MASTER IN COMPUTER SCIENCE

A Feature-based Configurtor for CAM

Colige, S

Award date:
2012

Awarding institution:
University of Namur

A Feature-based Configurator for CAM

Sarah Colige

Mémoire présenté en vue de l'obtention du grade de
Maître en Sciences Informatique
Année académique 2011 - 2012

Abstract

Scientific community is trying to create climate models as realistic as possible to simulate interactions and to analyse causal relationships in our environment. With these many simulations, researchers are trying to establish the most likely situations, and aim to predict future behaviours. These simulation models have a high rate of variability, in particular the choice of various elements that influence a particular simulation like the CO_2 cycle or the horizontal grid resolution. This variability is usually managed by a configurator. But the knowledge base used to develop these models is constantly evolving, which means a constant review of models and so the configurator.

In order to best manage the variability and the scalability supported by the configurator, we suggest to rely on a feature model that provides a reasoning and verification support that is easier to understand and edit than source code. The goal of our work is to apply this suggestion to a real case. This project uses a feature modelling language called Textual Variability Language (TVL) and implements a configurator based on this language for the configuration of atmospheric models by the Community Atmospheric Model (CAM).

Moreover, the confrontation to the considered real case shows the importance of default values to simplify the configuration of a model by the users. However these values are not part of feature models and require an extension of feature modelling languages. So we'll suggest a theoretical extension for TVL.

Keywords: Variability, Configuration, Feature modelling, Scalability, Default values

Résumé

La communauté scientifique essaie de créer des modèles aussi réalistes que possibles afin de simuler les interactions et d'analyser les liens de causalité au sein de notre environnement. Par ces nombreuses simulations, les chercheurs tentent d'établir les situations les plus probables, et ainsi visent à prédire les comportements futurs. Ces modèles de simulation présentent un haut taux de variabilité, notamment quant au choix des différents éléments qui influence une simulation particulière, comme le cycle de CO_2 ou la résolution de la grille horizontale. Cette variabilité est généralement gérée par un configurateur. Cependant la base de connaissance servant à l'élaboration de ces modèles est en constante évolution, ce qui implique une constante révision des modèles et du configurateur.

Afin de gérer au mieux la variabilité et l'évolutivité supportées par le configurateur, nous suggérons de se baser sur un modèle de variabilité offrant un support de raisonnement et de vérification plus facile à comprendre et à modifier qu'un code source. Le but de notre travail est d'appliquer cette suggestion à un cas réel. Ce projet utilise un langage de modélisation de la variabilité appelé Textual Variability Language (TVL) et implémente un configurateur basé sur ce langage pour la configuration de modèles atmosphériques par le Community Atmospheric Model (CAM).

De plus, la confrontation au cas réel étudié à montrer l'importance des valeurs par défaut afin de simplifier la configuration d'un modèle par les utilisateurs. Mais ces valeurs ne font pas partie intégrante des modèles de variabilité et nécessitent une extension des langages associés. Nous suggérerons donc une théorique extension pour TVL.

Mots-clefs: Variabilité, Configuration, Modélisation de la variabilité, Évolutivité, Valeur par défaut

Acknowledgements

I would like to thank my supervisor Professor Patrick Heymans for his guidance and his availability during the whole project. I would also like to express my gratitude for the opportunity he offered to me to work on an interesting subject in another university.

A special thanks goes to Professor Spencer Rugaber, Professor Leo Mark and Rocky Dunlap for their advices, their availability and their assistance during my internship. They form an amazing group of generous people and I really enjoyed to work with them.

I would also like to thank Arnaud Hubaux and Germain Saval who patiently read the several versions of my thesis and helped me to improve it.

Finally, I would like to thank all the people who encouraged me during the writing of this thesis and found appropriate words when I lacked of confidence.

Contents

List of Figures

Glossary

AST	Abstract Syntax Tree.
BDD	Binary Decision Diagram.
CAM	Community Atmospheric Model.
CCSM	Community Climate System Model.
CESM	Community Earth System Model.
CNF	Conjunctive Normal Form.
CSP	Constraint Satisfiability Problem.
DAG	Directed Acyclic Graph.
FBC	Feature-based Configuration.
FD	Feature Diagram.
FM	Feature Model.
FODA	Feature-Oriented Domain Analysis.
LKC	Linux Kernel Configurator.
NCAR	National Center for Atmospheric Research.
SAT	Boolean Satisfibility Problem.
SPL	Software Product Line.
SPLOT	Software Product Lines Online Tool.
SXFM	Simple XML Feature Model.
TVL	Textual Variability Language.

Chapter 1

Introduction

Climate change and impacts of our civilisation on the environment are the focus of many discussions of the scientific community, and the general public also has a growing interest about its future on our Earth. Therefore researchers, from Community Earth System Model (CESM) and others communities, are trying to create models as realistic as possible to simulate interactions and to analyse causal relationships in our environment. With these many simulations, researchers are trying to establish the most likely situations, and thus possibly help improve our fate. These simulation models have a high rate of variability, in particular the choice of various elements that influence a particular simulation like the CO2 cycle or the horizontal grid resolution. This variability is usually managed by a configurator. But the knowledge base used to develop these models is constantly evolving, which means a constant review of models and the configurator.

The high level of variability and the possibility to build several different models based on a common code base make CESM similar to a Software Product Line (SPL). Represent the variability of a family of related products like an SPL is generally done with feature modelling. Feature modelling appeared in 1980-1990 and was initially introduced to domain engineering by the Feature-Oriented Domain Analysis (FODA). This method uses a graphical notation to depict the relationship between the artefacts involved in the modelled system. Over the year, this kind of modelling progressed and language refinements appeared like attributes and cardinality-based decomposition. Tools has also been developed to provide a reasoning and automatic verification support. Currently, a text-based feature modelling language, called Textual Variability Language (TVL), is developed at the University of Namur. In this thesis, we use this language to model a real case from the climate community, that is Community Atmospheric Model (CAM). This case offered challenges, like highlighting options interactions, improving maintainability and supporting default values, that feature modelling should be able to meet, especially with text-based notation. We figured it out by implementing a feature-based configurator based on TVL. This configurator is only a prototype to support our study, but it helped us to well understand reasoning processes on Feature Models (FMs) with solvers.

During our work on this real case, CAM community, like some other industrial companies [20], expresses the need of default values to make the configuration process among the model easier for its users. Default values are equivalent to user's decisions that can be used to complete a configuration for example. They can be declaratively defined like constraints but the reader must pay attention to the fundamental difference between constraints and default values. Constraints restrict the relationship between artefacts modelled within FMs, while default values set the configuration when users don't make decisions. In addition, it came out that these default values usually take their values conditionally to

the state of other options in the model in order to refine their behaviour. However, this kind of structure is currently not supported by FMs in general. Only domain-dedicated configurators like Linux Kernel Configurator (LKC) [35] managed default values. Thus an extension of feature modelling languages should be produced. To do so, we introduce three views about default values and suggest a theoretical TVL extension. This suggestion contains the syntax and semantics to support these conditional default values.

Reader's guide This document is divided into two parts. First, the background explains the domain application of our study case, provides an overview of the feature modelling area and presents configuration management. Chapter 2 describes the foundation of CAM and states in details the configuration problem. Chapter 3 introduces SPLs, exposes the origins of FMs and describes TVL. Chapter 4 covers the management of configuration and summarises possible analyses that can be performed on FMs. This chapter also provides some configurators examples.

In the second part lays our contribution that is the implementation of a feature-based configurator and a proposal extension language for TVL. Chapter 5 describes in details the prototype of a feature-based configurator implemented during our internship and exposes how feature modelling and this kind of configurator could be helpful for CAM. Chapter 6 exposes possible integration views of default values into FMs and proposes a theoretical extension language of TVL to support these default values.

Finally, we conclude and propose future works in Chapter 7.

The work is followed by two appendixes. Appendix A provides the TVL feature model of CAM. Appendix B provides the syntax used by StringTemplate and describes the templates used to translate TVL into Simple XML Feature Model (SXFM) format and DOT.

Part I

Background

Chapter 2

Community Earth System Model

2.1 History[1]

The National Center for Atmospheric Research (NCAR) is a federally funded research and development center located in the United States. It is devoted to service, research and education in the atmospheric and related sciences. NCAR created the Community Climate Model (CCM) in 1983 as a freely available global atmosphere model especially for the whole climate research community. Over the past two decades, the formulation of the CCM has constantly improved, computers powerful enough to run the model have become relatively inexpensive and widely available, and usage of the model has become widespread in the university community, and at some national laboratories.

Until 1994, the original CCM did not include models of the global ocean and sea ice. According to this limitation, NCAR scientists planned to include models of the atmosphere, land surface, ocean and sea ice by developing and using a Climate System Model (CSM). The components were to be coupled without resorting to any "flux adjustments" (or flux correction) which adjust the surface heat, water and momentum fluxes artificially to maintain a stable control climate. Flux adjustments were used in early climate simulations when scientists discover that the model's predictions start to vary so much from the historical record that they have to go in and change the values inside the software to re-fit the model to what's actually happening. The plan of NCAR scientists was to focus initially on the physical aspects of the climate system, and then in a subsequent version to improve biogeochemistry and coupling to the upper atmosphere. The National Science Foundation (NSF), NCAR's primary sponsor, approved the plan, and the model development began immediately.

The first CSM Workshop was held in May 1996 in Breckenridge, Colorado. At this workshop the CSM components and the results of an early equilibrium climate simulation were presented. This workshop was also the first step for the full participation of the scientific community to develop CSM. A period of substantial organization progress has taken place since this 1996 workshop. A Scientific Steering Committee (SSC) has been formed to lead the CSM activity and working groups have been producing useful output. In addition to support from NSF, interest in the CSM from other agencies, like NASA, has developed. While working toward the second version of CSM, it was also *time to recognize the community of users and sponsors*, they changed the name of the model to the **Community Climate System Model (CCSM)**. The period since May 1996 has also been a time of substantial scientific progress. For example, a 300-year run has been

[1]This section summarizes the 'about' section of the official CESM website [32]

performed using the CSM, and the results from this experiment have appeared in a special issue of the Journal of Climate, 11, June, 1998.

Modelling climate's complexity[2].
The Figure 2.1.1, taken from a larger simulation of 20th century climate, depicts several aspects of Earth's climate system. Sea surface temperatures and sea ice concentrations are shown by the two color scales. The figure also captures sea level pressure and low-level winds, including warmer air moving north on the eastern side of low-pressure regions and colder air moving south on the western side of the lows. Such simulations, produced by the NCAR-based Community Climate System Model, can also depict additional features of the climate system, such as precipitation. Companion software, recently released as the Community Earth System Model, will enable scientists to study the climate system in even greater complexity.

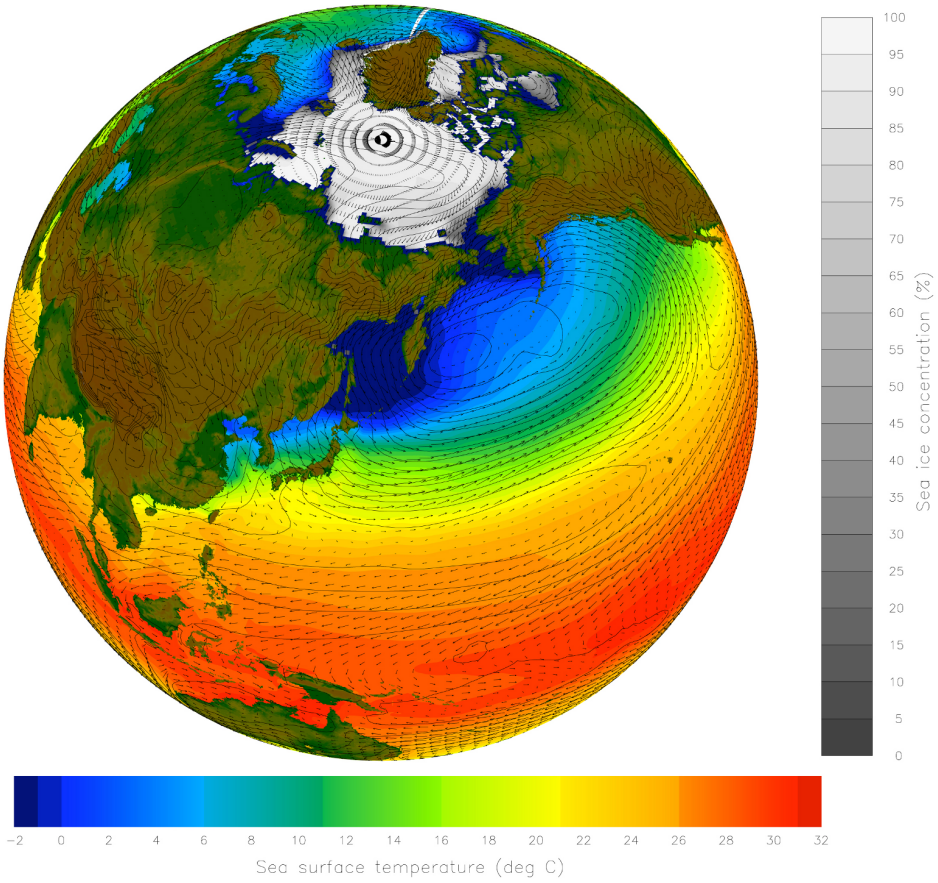


Figure 2.1.1: Simulation of 20th century climate (©UCAR. Image courtesy Gary Strand, NCAR)

Today, CESM is a "fully-coupled, global climate model that provides state-of-the-art computer simulations of the Earth's past, present, and future climate states."[32] CESM includes models of the atmosphere, land, land ice, ocean, and sea ice. These components are coupled with or without "flux adjustments" depending on the experiment.

[2]From the UCAR AtmosNews, New computer model advances climate change research, August 18, 2010 (`https://www2.ucar.edu/atmosnews/news/2366/new-computer-model-advances-climate-change-research`)

The most recent version CESM 1.0.3 was released on June 2011. Each release includes the complete collection of component model source code, documentation, and input data. The Figure 2.1.2 represents the software architecture of CESM. It is part of the poster[3] used by Kaitlin Alexander to present the results of a project in collaboration with Steve Easterbrook from the University of Toronto. Each component of the climate system has been assigned a colour: 1. atmosphere - *Purple*; 2. ocean - *Blue*; 3. land - *Orange*; 4. sea ice - *Green*; and 5. land ice - *Yellow*. Model code for a component is represented with a bubble. Fluxes are represented with arrows, in a colour showing where they originated. Couplers are grey. Components pass fluxes through the coupler. The area of a bubble represents the size of its code base, relative to other components in the model. Radiative forcings are passed to components with plain arrows.

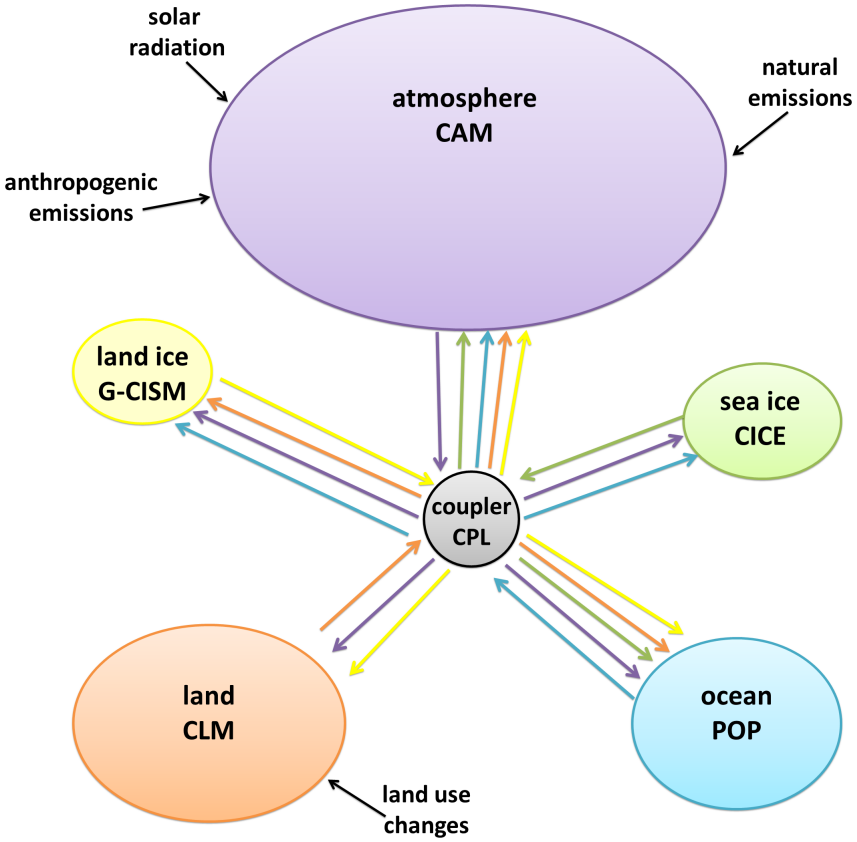


Figure 2.1.2: Software architecture of CESM

CESM researchers anticipate many important changes in the climate modelling enterprise over the next five years. Computer power will increase and be able to support more elaborate and more sophisticated models and modelling studies, using increased spatial resolution and covering longer interval of simulated time. Understanding of many component processes represented in CESM will be improved, including cloud physics; radiative transfer; atmospheric chemistry, including aerosol chemistry, boundary-layer processes, polar processes, and biogeochemical processes; and the interactions of gravity waves with the large-scale circulation of the atmosphere. Understanding of how these component processes interact, numerical methods for the simulation of geophysical fluid dynamics and observations of the atmosphere, including major advances in satellite observations will

[3] Available at `http://climatesight.org/2011/12/14/the-software-architecture-of-global-climate-models/`

also increase.

2.2 Purpose[4]

The CESM project concerns important areas of climate system research. Especially, it is aimed at understanding and predicting the climate system. Changes in climate, whether anthropogenic or natural, involve a complex interplay of physical, chemical, and biological processes of the atmosphere, ocean, and land surface. The challenges of modelling the roles of anthropogenic emissions of carbon dioxide, reactive trace gases, and of changing land use in the earth system require a coupled-climate-system approach where components interact and influence each other. While an appreciation that land-ocean-atmosphere interactions influence climate is not new, the emergence of coupled-climate-system questions as central scientific concerns of geophysics constitutes a major change in the research agendas of atmospheric science, oceanography, ecology, and hydrology.

A comprehensive CESM accurately represents the principal components of the climate system and their couplings. And its development requires both wide intellectual participation and computing capabilities. Therefore, CESM must include an improved framework for coupling existing and future component models developed at multiple institutions, to permit rapid exploration of alternate formulations. This framework must be flexible to components of varying complexity and at varying resolutions, in accordance with a balance of scientific needs and resource demands. In particular, CESM must accommodate an active program of simulations and evaluations, using an evolving model to address scientific issues and problems of national and international policy interest.

In the long-term, the CESM project has the following ambitious goals, as listed on the official web site of CESM:

- to develop and continually improve a comprehensive CESM that is at the forefront of international efforts in modelling the climate system, including the best possible component models coupled together in a balanced, harmonious modelling framework;
- to make the model readily available to, and usable by, the climate research community, and to actively engage the community in the ongoing process of model development;
- to use CESM to address important scientific questions about the climate system, including global change and interdecadal and interannual variability; and
- to use appropriate versions of CESM for calculations in support of national and international policy decisions.

Even if the CESM project remains focused on comprehensive climate modelling, efforts using simplified models are also important and complementary. Therefore, this kind of efforts is undertaken by many individuals, including some CESM participants. Actually, CESM is developed thanks to the active work of relatively small teams of scientists called the CESM Working Groups. These groups work on individual component models or specific coupling strategies. Each team takes responsibility for developing and continually improving its component of CESM consistently with the CESM goal of a fully-coupled model and with the CESM design criteria. Each team will decide their own development

[4]This section summarizes information available at the official CESM website [32]

priorities and work schedules, consistent with the overall goals of CESM, and subject to oversight by the CESM Scientific Steering Committee (SSC) that leads the CESM activity.

The following Table 2.1 from the official website of CESM lists the CESM working groups, briefly describes them and names their liaison agent(s).

Table 2.1: List of the CESM working group, their brief description and liaison agent(s).

Working Group	**Liaison**	**Description**
Atmosphere Model	Cecile Hannay	The Community Atmosphere Model (CAM) is the latest in a series of global atmosphere models developed at NCAR for the weather and climate research communities. CAM also serves as the atmospheric component of CESM.
Land Ice	Stephen Price	The Land Ice Working Group (LIWG) was formed to develop and apply the ice sheet model component of CESM and to assess the role of land ice in climate change and sea-level rise.
Land Model	Sam Levis	The Community Land Model is the land model for CESM and the Community Atmosphere Model (CAM).
Ocean Model	Susan Bates	The Ocean Model Working Group was formed to discuss issues related to ocean components in the CESM context and to coordinate the development and testing of ocean components.
Polar Climate	David Bailey	The Polar Climate Working Group (PCWG) is a consortium of scientists who are interested in modeling and understanding the climate in the Arctic and the Antarctic, and how polar climate processes interact with and influence climate at lower latitudes.
Biogeochemistry	Keith Lindsay	The overall goal of the Biogeochemistry Working Group (BGCWG) is to improve our understanding of the interactions and feedbacks between the physical climate and biogeochemical systems under past, present and future climates.
Chemistry-Climate	Simone Tilmes	The CESM chemistry-climate working group is formed to focus on the coupling between the climate system, aerosols, atmospheric composition and chemistry.
Climate Variability & Change	Gary Strand and Adam Phillips	The Climate Variability & Change Working Group (CVCWG) was created from a merge of the Climate Variability and Climate Change Working Groups in August 2011. CVCWG conducts simulations with CESM and its component models that are available to the broad research community.

continued on next page

continued from previous page

Working Group	Liaison	Description
Paleoclimate	Nan Rosenbloom and Christine Shields	The largest climate changes that have occurred on Earth, such as the Ice Ages, are those recorded in the geologic record. Understanding the causes of such past climate changes is an essential part of developing and validating models of future climate change.
Societal Dimensions	–	The overall goal of the Societal Dimensions Working Group is to enhance CESM and its application in order to improve understanding of the interactions between human and earth systems.
Software Engineering	–	The Software Engineering Working Group (SEWG) was formed to examine software engineering issues related to creating and executing CESM model codes, and to recommend and help coordinate software engineering practices that will help the CESM project to achieve its goals.
Whole Atmosphere	Mike Mills	The Whole Atmosphere Working Group (WAWG) was formed to examine the range of altitude from the Earth's surface to the thermosphere.

2.3 Community Atmosphere Model[5]

The CAM is the latest in a series of global atmosphere models developed at NCAR for the weather and climate research communities. CAM also serves as the atmospheric component of CESM. The most recent version is 5.1, it is used both as a standalone model and as the atmospheric component of CESM. The standalone model usage means that the atmosphere model is coupled to an active land model (CLM), a thermodynamic only sea ice model (CICE), and a data ocean model (DOCN). Researchers speak of "doing CAM simulations" in that case. When CAM is coupled to active ocean and sea ice models then they refer to the model as CESM.

This atmosphere model is designed to manage more than seventy options like the type of the physical and chemical components or the grid size, with dependencies and exclusions between options. A simulation model can be configured thanks to a specific script. This script is the current representation of "expert knowledge", it represents what configuration possibilities are available and high-level constraints among them. The configuration is designed with the criterion that the user is the expert, so the user is allowed to make the choices that he wants under the assumption that he knows what he is doing. But a design goal is that if the configuration succeeds then the build should succeed.

The primary mechanism for representing and disseminating validated CESM is via component sets or "compsets", the same applies for CAM. A compset is a particular mix of components, along with component-specific configuration and/or namelist settings. Quality is ensured by routinely testing the supported compsets which have a low granularity and were rigorously validated. A user can choose to execute a run with custom changes, but he will be responsible for validating the results. A typical way of doing this is by comparing those results with output from a validated compset.

[5]This section summarizes the 'Community Atmosphere Model (CAM)' section of the official CESM website [31]

2.4 Sample of CAM experiment result

This section presents a sample of the output data computed with CAM5.0 in the standalone mode for a simulation from 1980 to 1999 at 2 degree resolution. The conventional name of this experiment is `f40_amip_cam5_c03_78b`. Output data are interpreted by a diagnostics packages that produces a number of diagnostic plots and metrics in a series of easily navigable web pages. The following diagnostics are part of atmosphere diagnostics[6].

The Figure 2.4.1 represents the annual implied northward heat transport for each ocean. The red curve is the result of the simulation and the black curve is based on observed data from the National Centers for Environmental Prediction (NCEP). We can see that even if curves are not exactly the same, they follow the same trend.

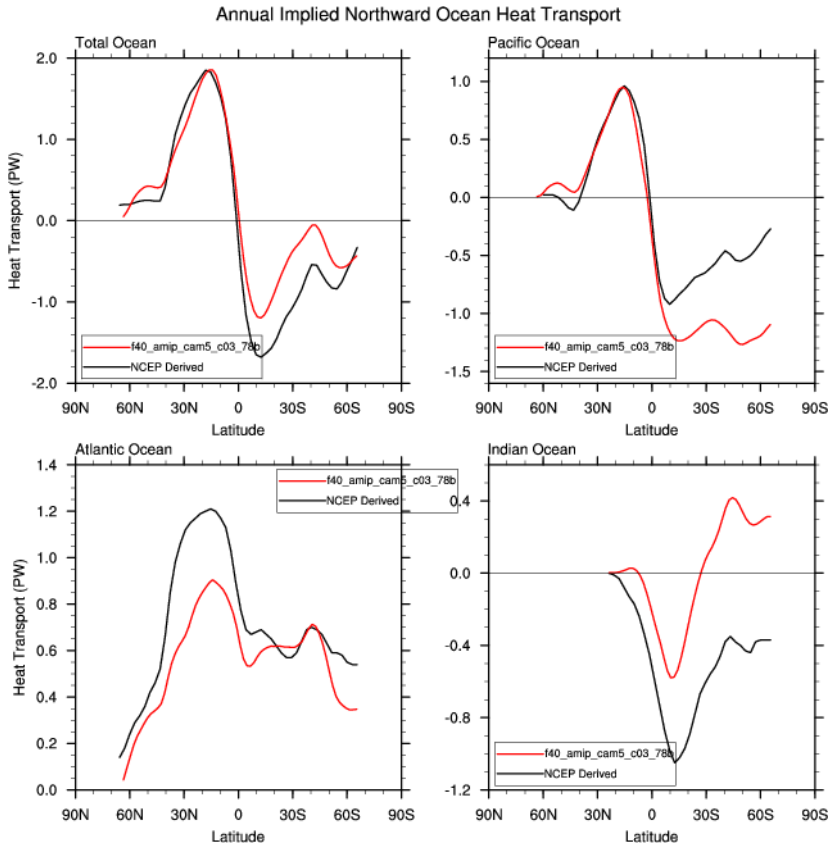


Figure 2.4.1: Annual Implied Northward Ocean Heat Transport

The Figure 2.4.2 represents the 2-meter air temperature over the world (horizontal vector plots of DJF). The first graph shows the temperatures computed by the simulation between 1980 and 1999. The second one shows the temperatures from Legates and Willmott data set. The third one shows the temperature difference between CAM data and Legates data. We can see that around the equator temperatures are similar and when approaching the poles a gap appears.

2.5 CAM configuration

CAM has a configuration process designed to support a number of different scientific scenarios. Because of this flexibility, the configuration of CAM is a complex process. To

[6]Simulation reference: f40_amip_cam5_c03_78b. Available at the official website of CESM [32].

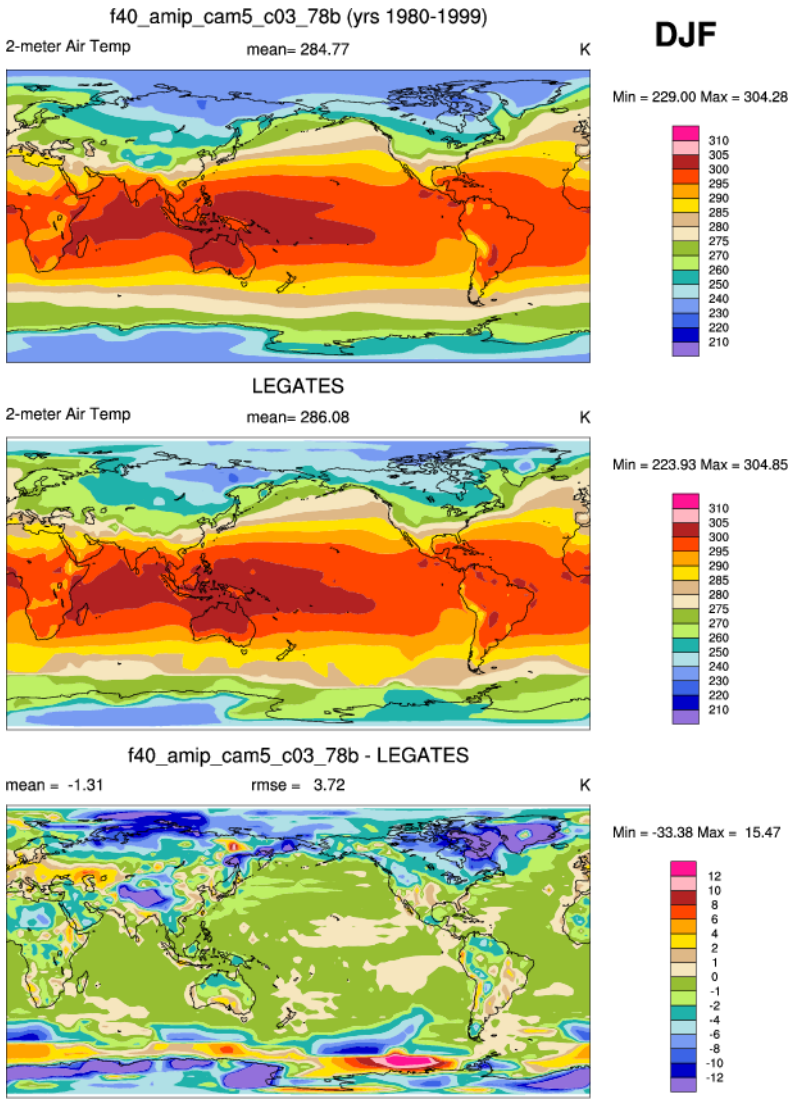


Figure 2.4.2: 2-meter air temperature over the world

build and run CAM in standalone mode the following is required:

- The source tree. CAM-5.1 is distributed with CESM-1.0.3.
- Perl (version 5.4 or later).
- A GNU version of the **make** utility.
- Fortran90 and C compilers.
- A NetCDF library (version 3.6 or later) that has the Fortran APIs built using the same Fortran90 compiler that is used to build the rest of the CAM code. This library is used extensively by CAM both to read input datasets and to write the output datasets.
- Input datasets. The required datasets depend on the CAM configuration.

When all required items are available, building and running CAM takes place in the following steps:

1. **Configure model.**
   This step is accomplished by running the configure utility (in this document we will call it configuration script) and providing a number of command line parameters that describe his particular configuration choices including the machine definition, the climate model options to use and archiving options. The user can supply only parameters that differ from the default values of most configuration script parameters. Only few compile-time parameters such as the dynamical core (Eulerian Spectral, Semi-Lagrangian Spectral, Finite Volume, or Spectral Element), horizontal grid resolution, and the type of parallelism to employ (shared-memory and/or distributed memory) has no default value and must be supplied.

2. **Build model.**
   This step includes compiling and linking the executable using the GNU make command (gmake). configuration script creates a `makefile` in the directory where the build is to take place. Then the user needs only execute the gmake command in this directory.

3. **Build namelist.**
   This step is accomplished by running the build-namelist utility, which supports a variety of options to control the run-time behavior of the model. Any namelist variable recognized by CAM can be changed by the user via the build-namelist interface. The user can also specify a set of namelist variable settings for running particular types of experiments.

4. **Execute model.**
   This step includes the actual invocation of the executable. When running using distributed memory parallelism this step requires knowledge of how one's machine invokes (or "launches") Message Passing Interface (MPI) executables. When running with shared-memory parallelism, using Open Multi-Processing (OpenMP), one may also set the number of OpenMP threads. On most HPC platforms access to the computing resource is through a batch queue system.

In this thesis, we will focus on the configuration step. A primary function of the CAM configuration script is to enforce **constraints** among the configuration options. The configurable nature of CAM makes it similar to a SPL. An SPL is a collection of similar software systems that create from a shared set of software assets using a common means of production. Carnegie Mellon Software Engineering Institute defines an SPL as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."[39] In other words, the CAM codebase can be used to create a large number of related software applications that vary in different ways.

Currently, CAM uses a configuration script in Perl that is the current representation of *expert knowledge*. It represents what configuration possibilities are available and high-level constraints among them. So, configuration script checks constraints among the configuration options and selects appropriate source code directories to finally produce a `makefile`[7] and a `filepath`[8] according to the parametrization. But this parametrization must be done by a domain expert familiar with the configuration script of CAM, otherwise he could receive a series of error messages.

[7] Makefile is used by **Make** utility to automatically build executable programs.
[8] Filepath contains the list of source directory paths required to compile with **Make**.

2.6 Challenges

The current configuration process is entirely managed by a script in PERL. That may cause some difficulties for different points what we list here (some of the following points may overlap):

C1: Improving maintainability of CAM system.
Source code like scripts has a significant risk of becoming inconsistent as systems evolve. The best simple example of inconsistency is conflicting constraints like an exclusive constraint and a required constraint on two options. If the system covered by the script has a lot of constraints, it can be really difficult to find and understand the meaning of a particular constraint and its influence on other ones. Currently, constraints between CAM options like their dependencies or their exclusions, are defined in the configuration script as an if-condition. And sometimes assuming the result of a previous constraint, thanks to the sequential execution of the script. But what would happen if, evolving, one assumption is no longer respected and is fundamental for an other constraint?

C2: Setting up a separation of concerns.
The configuration script manages three types of parameters: 1. the machine definition; 2. the climate model options to use; and 3. archiving options. We can assume that users want to make the difference between these parameters, but currently they are just listed and sometimes mixed inside the script. configuration script is the definition of CAM models through constraints and the tool to check the consistency of user's choices. Moreover, the configuration script can also be used under two points of view: the programmer view and the user view. Indeed, the same script is used to debug the configuration process and to configure CAM models.

C3: Highlighting options interactions.
CAM has its documentation where each parameter is described and listed both in a namelist and in the 'help' display of the configuration script. But the support for certain features and the effect of making a particular configuration choice are unclear. Users may be confused about how configuration options interact with each others, what may lead to a trial-and-error approach to configuration. Even in this way, users are not sure about the application of their choices. This lack of documentation appeared thanks to an analysis performed by Rocky Dunlap on the CESM forum[9].

C4: Supporting default options.[10]
Configuration of Climate Model simulation is a highly complex task due to the need to support a large number of scientific scenarios, introducing a large number of parameters and options to the configuration setup. Specifying all possible parameters and configuration options is impractical and time-consuming for users. One of the ways CAM) counters this complexity is by the introduction of *defaults options*, a set of selected options that are already configured to work together. The introduction of default options helps reduce the number of option needed to be selected, but the relationships and the constraints between options remains opaque to the user.

[9] CESM Forum Analysis – Scientific Configuration on Rocky's Blog at `http://rockydunlap.wordpress.com/2010/10/26/cesm-forum-analysis-scientific-configuration/`

[10] Based on the problem statement of Sameer's Research [3]

C5: Increasing accessibility.
CAM, and more generally CESM, is designed to be configured by expert users. As a design criterion is "the user is allowed to make the choices that he wants under the assumption that he knows what he is doing". But here, expert users point out a subset of climate experts whom can understand the output data set, analyse it and validate or invalidate it. This subset is restricted to users familiar with the configuration script. A new user must start its CAM usage by learning how the configuration script works, instead of just making choices according to his domain knowledge and what he wants experiment.

Actually, *"scientist who runs the models as part of their research often has junior scientist working as "configuration manager" who would become the local experts for knowledge of how to configure models for particular runs. [...] This complexity in model configuration slows down scientific progress."* [3]

C6: Supporting domain and model evolution.
Although CAM can be configured to produce a variety of different software products, it is not accurate to call it an SPL (at least in the traditional sense of the term) because it is generally much less stable than a typical SPL. CAM is more like a research tool that is constantly under development. There are many ways to configure CAM, but also many ways that it can be misconfigured.

Doing research with CESM *(and CAM)* is analogous to working in a physics lab. It is an experimental environment that is never static.

As CAM can be *thought as* an SPL, we can try to apply feature modelling which is widely used in SPL engineering to represent the variability. FMs offer a convenient and simple notation for documenting commonality and variability in SPL. FMs also allow one to describe more or less simple constraints among the features. Moreover, feature modelling is supported by mature tools for configuration and various analyses, they will be further detailed in Chapter 4. This should be helpful, at least, for visualizing the variation points (configuration options) of CAM and constraints among the different configuration options [14]. This modelling can help us take up the challenges described previously.

Chapter 3

Feature Modelling

To represent a family of related programs, like an SPL, feature modelling is generally used. FMs are an adequate compact representation of all the products of SPLs in terms of "features". They are widely used during the whole product line development process and are commonly used as input to produce other assets such as documents, architecture definition, or code. When the units of program construction are features, every program in an SPL is identified by an unique and legal combination of features, and vice versa. FMs clarify the constraints between different features provided by a system. Constraints can be represented as a relationship between a parent feature and its children (sub-features) or expressed with additional cross-tree logical constraints. [25, 27]

Most of the time, a graphical notation based on FODA[23] is used to represent the variability in an SPL. But this approach lacks scalability and becomes a burden for large FMs. A text-based feature modelling notation can be more appropriate and provides engineers with a comprehensive language supporting large-scale models if it provides a modularisation mechanism. Such a text-based notation, TVL, is being developed by the University of Namur [10]. TVL covers most of the constructs of existing feature modelling languages, including cardinality-based decomposition and features attributes while staying light.

3.1 Software Product Lines

SPL refers to software engineering methods, tools and techniques for creating a collection of similar software systems from a shared set of software assets using a common means of production. Carnegie Mellon Software Engineering Institute defines an SPL as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [39].

3.1.1 Concepts[1]

SPLs are used to manage variability inside software productions when reuse is predicted in one or more products in a well defined product line. This variability is represented by software artefacts, aka. *core assets*. According to Krueger [26], SPLs can be described in terms of four simple concepts, as illustrated in the Figure 3.1.1:

[1]Based on the 'Introduction to Software Product Lines' by C.W. Krueger [26]

Software asset inputs : a collection of software assets (such as requirements, source code components, test cases, architecture and documentation) that can be configured and composed in different ways to create all of the products in a product line. Each of the assets has a well defined role within a common architecture for the product line. To accommodate variation among the products, some of the assets may be optional or have internal *variation points* that can be configured in different ways to provide different behaviour.

Decision model and product decisions : the *decision model* describes optional and variable features for the products in the product line. Each product int the product line is uniquely defined by its *product decisions* (choices for each of the optional and variable features in the decision model).

Production mechanism and process : the means for composing and configuring products from the software asset inputs. Product decisions are used during production to determine which software asset inputs to use and how to configure the variation points within those assets.

Software product outputs : the collection of all products that can be produced for the product line.

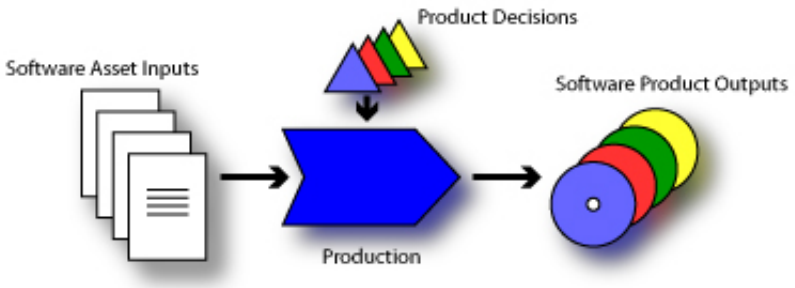


Figure 3.1.1: Basic Software Product Line Concepts [26]

The main objectives of SPLs are illustrated through these concepts. SPLs aims "to capitalize on commonality and manage variation in order to reduce the time, effort, cost and complexity of creating and maintaining a product line of similar software systems." [26] The **capitalization on commonality**, thanks to consolidation and sharing within the software asset inputs, allows to avoid duplication and divergence. The **variation management** makes the location, rationale, and dependencies for variation explicit thanks to a clear definition of the variation points and decision model.

The primary specificity of SPLs is the presence of variation in some or all of the software assets. At the beginning of a SPL lifecycle, software assets contain variation points that represent unbound options about how the software will behave. Later during the production process, a selection among the options of each variation points utilizes product decision, in order to fully-specify the behaviour of the variation point in the final product. The time at which the decisions for a variation point are bound is referred to as the *binding time*.

Examples of different binding times for SPLs include, listed by Krueger [26]:

- *Source reuse time.* Decisions bound when reusing a configurable source artefact;

- *Development time.* Decisions bound during architecture, design, and coding;
- *Static code instantiation time.* Decisions bound during assembly of code just prior to a build;
- *Build time.* Decisions bound during compilation or related processing;
- *Package time.* Decisions bound while assembling binary & executable collections;
- *Customer customizations.* Decisions bound during custom coding at customer site;
- *Install time.* Decisions bound during the installation of the software product;
- *Startup time.* Decisions bound during system startup;
- *Runtime.* Decisions bound when the system is executing.

Multiple binding times can be utilized in a software product line. This allows some decisions to be bound earlier in the lifecycle and other decisions to be deferred until later in the process. For example, some decisions should be made by a product manager at the company developing the software, while other decisions should be made by the end customer that will use the software. In other words, Kruger says:

> With multiple binding times, the software product outputs from binding decisions at one production stage become *partially instantiated* software asset inputs for binding decisions at the next production stage. The Figure 3.1.2 illustrates two binding times, though more are possible. [26]

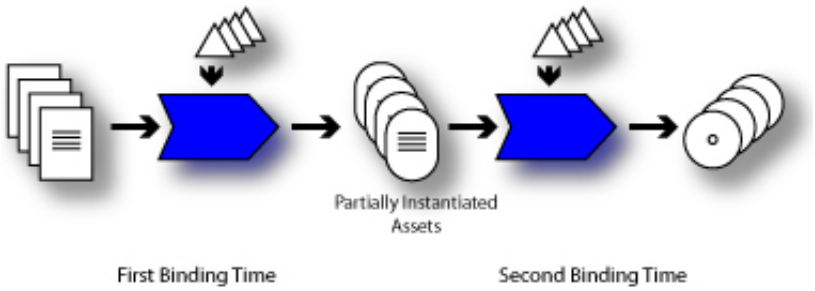


Figure 3.1.2: Multiple binding times [26]

As bind variations in asset are determined through product decisions, partially or fully instantiated product outputs can be created by the production operation from software asset inputs. Production in SPLs can be fully *automated*, completely manual, or somewhere in between. Application generators and product *configurators* are a good example of a fully automated production approach. In this approach product decisions provide sufficient information to automatically generate the product outputs. An example of a completely manual approach is a textual production plan. In this case, software engineers interpret and follow directions in the plan and the product decisions to tailor, integrate, and provide some code to link the software assets in order to create products.

3.1.2 Benefits & Disadvantages[2]

SPLs offers some benefits, like the following:

[2]Based on the master's thesis of P. Faber[17] and the 'Introduction to Software Product Lines' by C.W. Krueger [26]

B1: **Reduction of development costs** through the reuse of the artefacts, the cost of each artefact is inferior to their cost if they are developed by the classical way.

B2: **Reduction of time-to-market.** In general and for a large production, the time required to develop a new product based on artefacts is shorter as the common artefacts are already developed and available thanks to older products.

B3: **Reduction of complexity** through the variability management and an efficient reuse of artefacts according to well defined processes and plans that simplify the software conception.

B4: **Improvement of the output software quality** through the reuse of artefacts, they are widely tested, so a bug inside an artefact can more probably be detected and corrected.

However, SPLs require more effort to be implemented and some factors like the following can compromise it:

D1: **Low number of products.** Benefits of SPLs appear for a certain number of developed products. Create an SPL for too less products may lead a company to never cover its investment, if the development cost stays superior to the benefits.

D2: **High variability between products.** If the product family is too diversified, the development of enough flexible artefact may generate additional costs. Moreover, the range of reused artefacts should be low and reuse advantage would be lost. And finally, the development cost of each product may become higher than the cost of the same product developed by the traditional way.

D3: **Lack of knowledge about the domain** Create an SPL requires to well know the domain (technologies, market, clients,...). The company must be able to determine the product set while taking account of the future needs of its clients. Otherwise, the SPL should be updated to add new products what generates additional costs. Domain engineers and software engineers must also well know the domain to design and implement the variability. Wrong choices may generate products that don't meet the requirements. Each actor must know the domain, or the conception of the SPL may fail.

D4: **Domain instability** If the domain of the SPL is unstable or evolve too quickly (e.g., once every 6 months), the investments for the maintenance of the SPL may never pay back. For each update, artefact may be changed and the benefits earn with the products could always be lower than the cost of these updates.

3.2 Feature Oriented Diagram Analysis[3]

FODA is a domain analysis method. It was first developed by the Software Engineering Institute, in 1990, as a comprehensive analysis and refinement of technology developed from 1983-1990. While some aspects of FODA have changed, and it has become integrated with model-based software engineering, FODA is still known as the method that initially introduced feature modelling to domain engineering. [13]

[3]Based on the initial technical report of FODA [23]

FODA intents to support functional and architectural reuse. Its objective is to create a domain model which represents a family of systems which can then be refined into the particular desired system within the domain. In the initial technical report [23], Kang *et al.* introduced the concept of the FMs and defined a feature as "a user-visible aspects or characteristic of the domain". The features define both common aspects of the domain as well as differences between related systems in the domain. Then the FM is a better communication medium since it provides this external view that the user can understand.

3.2.1 Feature Analysis

The FM is produced during the first activity of FODA domain modelling phase, called "feature analysis". The purpose of this activity is to capture in the model the end-user's understanding of the general capabilities of applications in a domain. The approach has an end-user perspective of the functionality of applications. It focuses on the "services" provided by the applications and the operating environments in which the applications run. Features are the attributes of a system that directly affect end-users. The end-users have to make decisions regarding the availability of features in the system.

According to Kang et al. [23], the components of the FM are as follows:

- *Feature diagram:* A graphical And/Or hierarchy of features
- *Composition rules:* Mutual dependency (Requires) and mutual exclusion (Mutex-with) relationships
- *Issues and decisions:* Record of trade-offs, rationales, and justifications
- *System feature catalogue:* Record of existing system features

The following paragraphs discuss each of these parts of the FM in detail.

Feature Diagram

The Feature Diagram (FD), shown in Figure 3.2.1, is an And/Or tree of different features. *Optional* features are designated graphically by a small circle immediately above the feature name, as in Air conditioning. *Alternative* features are shown as being children of the same parent feature, with an arc drawn through all of the options, as is the case in Transmission. The arc signifies that one an only one of those features must be chosen. The remaining features with no special notation are all *mandatory*. The line drawn between a child feature and its parent feature indicates that a child feature *requires* its parent feature to be present; if the parent is not marked as valid[4], then the child feature for that system is in essence "unreachable."

Composition Rules

A composition rule constraints the use of a feature. It has two forms: 1. one feature *requires* the existence of another feature (because the first depends on the other); and 2. one feature is *mutually exclusive with* another (they cannot coexist). The textual representation for these rules is as follows:

$$< \text{feature1} > (\text{`requires'} | \text{`mutex-with'}) < \text{feature2} >$$

An example of a composition rule used in the vehicle domain is:

$$\text{Air conditioning } \textit{requires } \text{Horsepower} > 100$$

[4] A feature is marked as "valid" if it is either: 1. marked "valid"; 2. mandatory; 3. *not* marked "invalid"; or 4. *required* by a "valid" feature. (Section 3.2.2)

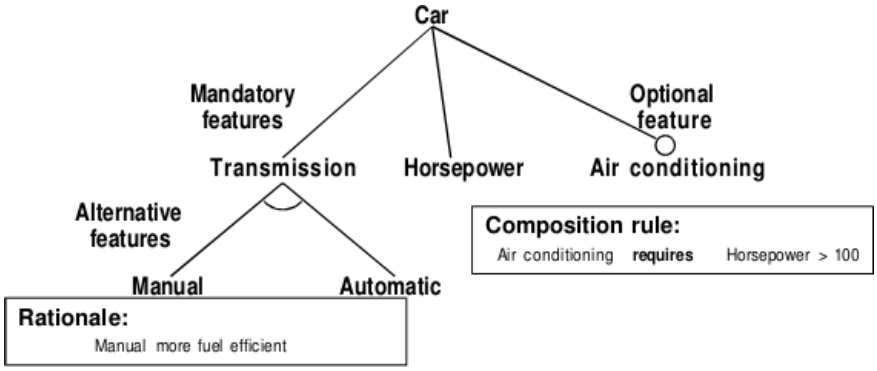


Figure 3.2.1: Example Showing Features of a Car

Issues and Decisions

A record of the issues and decisions that arise in the course of the feature analysis must be incorporated into the FM to provide the rationale for choosing options and selecting among several alternatives. As an example, the Transmission feature has two different alternatives: Manual and Automatic. It is impossible to select one or the other without having access to the same information the original designer had. However, if the FM contains a record of the original rationales, it is a simple process. An issue is composed of a short description, a raising point and a list of feature decisions. A decision has a description and a rationale.

System Feature Catalogue

Experience with existing systems in the domain is a useful source to gather information for the domain analysis. Record the features and feature values of actual existing systems (even in the case of manual methods) is important to allow later modelling of the systems in terms of their features.

3.2.2 Automated Tool Support for Features[5]

Manually creating a FM that correctly describes a complex domain is a large effort and validating that model in some way is still more difficult. Because the FODA method was new, and no existing automated tool support was available, a prototype tool was developed using Prolog. The primary function of the tool is to validate the usefulness of the feature analysis approach, and secondarily to establish some baseline requirements for future automated support for the method. As the tool is separated from the information about the domain being analysed, it may be applied to any domain. The features are stored in a Prolog fact base, along with the composition rules and other related information. The tool supports definition of existing or proposed systems by allowing arbitrary sets of feature values to be specified and checked. The composition rules relating the features are enforced, as are standard rules about completeness of the model. Given a set of user-specified (i.e. "marked") features, the automated features tool presently performs the following functions:

- Checks for *all* features that are specified, but which may not be *reachable*.
- Marks a feature as "valid" if it is either:

[5]The content of this sub-section comes from the FODA technical report [23, Section 7.3.2.6], only few adaptations was performed.

  - marked "valid",
  - mandatory,
  - *not* marked "invalid", or
  - *required* by a "valid" feature.

- Marks a features as "invalid" if it is mutually exclusive with a "valid" feature.
- Produces an error if a feature is marked as both "valid" and "invalid."
- Enforces the proper selection of alternatives:
  - at least one alternative *must* be marked "valid."
  - more than one alternative *cannot* be "valid."

3.2.3 Limitations[6]

Composition Rules vs. And-Or of Features

Hierarchical relationships between the features in the and/or tree are an alternate graphical representation of the *requires* composition rule. Use two alternate representations of the same relationship complicates the FM, but the FD provides a way for users of the model to "see" some of the feature relationships. That is difficult to do with a model composed solely of composition rules. Sophisticated automated support for the interactive display of the FM, such as hypertext techniques, could provide displays that would make the FD redundant and hence unnecessary.

Manual vs. Automated Methods

As the amount of information needed to describe a domain grew, the manual techniques became more complex. For example, the FD had no way of displaying the effects of the composition rules on the relations between features to show that the existence of one feature was conditional on another feature. To handle this some notational extensions to the diagram were tried, but these only made an already complex diagram larger and more abstruse. The FD had been split across several pages along arbitrary boundaries in an attempt to make it more manageable, and contained inconsistencies that could be found only through exhaustive manual examination. This situation was a primary reason for building a prototype automated features tool to represent the FM and support consistency and completeness checking.

3.3 Text-based Variability Language[7]

Most authors used a graphical notation based on FODA. The main drawback of those approaches is their lack of scalability: they generally do not fit real-size problems. Indeed, its graphical syntax does not account for attributes or complex constraints and becomes a limitation for large FMs. So Pr. Heymans and his staff at PReCISE Research Center [2] develop TVL.

TVL is a text-based feature modelling notation with a C-like syntax. Objectives of the language are to be *scalable* and comprehensive. TVL is light and offers modularisation

[6]The content of this sub-section is extracted from the FODA technical report [23, Section 8.1], only few adaptations was performed.

[7]Based on the TVL specification [12] and the "Introducing TVL" paper [10] available at the TVL website (`http://www.info.fundp.ac.be/~acs/tvl`)

mechanisms to provide engineers with a human-readable language supporting large-scale models. TVL also covers most of the FM dialects proposed in the literature, including cardinality-based decomposition and feature attributes. Furthermore, TVL can serve as an extensible storage format for feature modelling tools. TVL is currently a language proposal. It is formally defined with an LALR grammar, a formal semantics and a reference implementation in Java is available at the TVL website[8].

3.3.1 Concepts[9]

In this sub-section we present an overview of the TVL syntax using snippets. The following paragraphs introduce five major parts of the language: feature, attributes, expressions, constraints and modularisation mechanisms. The different concepts of TVL will be illustrated using a basic personal computer product family example FD visible in Figure 3.3.1. The *Computer* consists of a *Motherboard*, a *CPU*, a *Graphic Card* and some *Accessories*, which are optional (indicated by the hollow circle); all of these features are then further decomposed. In addition, although not shown in the figure, each of the features has a *price*, which can be modelled as an attribute, a typed parameter attached to each feature [7, Section 2.2].

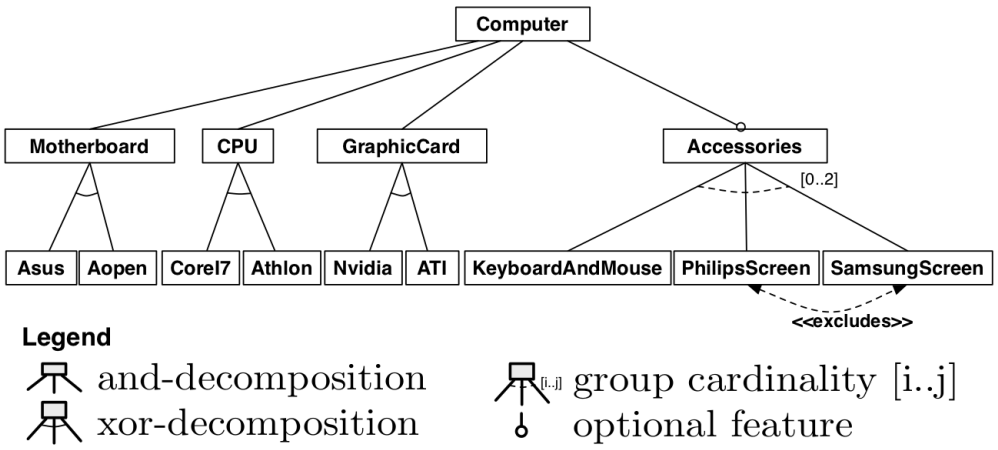


Figure 3.3.1: Computer example FD [12]

Feature hierarchy

The TVL language has a C-like syntax: it uses braces to delimit blocks, C-style comments, semicolons to delimit statements. The authors' rationale for this syntax choice is that nearly all computing professionals have come across a C-like syntax and are thus familiar with this style. Furthermore, many text editors have built-in facilities to handles this type of syntax. In the example, the root feature, *Computer*, is decomposed into four sub-features by an *and*-decomposition: *Motherboard*, *CPU*, *GraphicCard* and *Accessories*. Furthermore, the *Accessories* feature is optional while the other three features are mandatory. In TVL, this is written as follows:

```
1  root Computer{
2    group allOf{
3      Motherboard,
```

[8] http://www.info.fundp.ac.be/~acs/tvl

[9] The content of this sub-section is extracted from the TVL specification [12, Section 2], only few adaptations was performed.

```
4       CPU,
5       GraphicCard,
6       opt Accessories
7     }
8   }
```

A decomposition type in TVL is defined with the **group** keyword. Predefined decomposition operators are **allOf**, as used in this example for an *and*-decomposition, **someOf** for *or*-decompositions and **oneOf** for *xor*-decompositions. It is also possible to specify a cardinality-based decomposition with the **group [i..j]** syntax, where i and j are the lower and the upper bounds of the cardinality. When defining a cardinality, one can use the asterisk character * to denote the number of children in the group, for instance **group [1..*]** would be equivalent to **group someOf**. This way, the engineer does not have to update the cardinality each time the number of children changes. Optional features like *Accessories* are declared by putting the **opt** keyword in front of their name.

FMs most commonly have a tree structure but, sometimes a Directed Acyclic Graph (DAG) structure - a feature can have several parents - might be useful [24]. DAG structures can also be modelled in TVL with the **shared** keyword associated to a feature name. This means that the shared feature has several parents as it is illustrated int he following example where feature D has features B and C as parents:

```
1  root A
2    group oneOf{
3    B group allOf {D},
4    C group allOf {shared D}
5    }
```

Attributes

In the example, the *Motherboard* has four attributes: a price, a width, a height and a socket type. TVL supports four different attribute types: integer (**int**), real (**real**), Boolean (**bool**) and enumeration (**enum**). In the example, *price*, *width* and *height* are integers. Furthermore, the *price* value is limited to values between 0 and 500. In TVL, this is expressed as follows:

```
1  Motherboard {
2    int price in [0..500];
3    int width;
4    int height;
5  }
```

Attributes are thus declared by defining their type and name inside the definition block of the feature they belong to. Each attribute declaration is terminated by a semicolon. The **in** keyword is optional, it can be used to restrict the domain of an attribute (which might speed up automated reasoning). When declaring an attribute as an enumeration type, this means that it will take exactly one of a set of predefined values. The *socket*, for instance, is either *LGA1156* or *ASB1*. We thus declare it as an enum.

```
1  Motherboard{
2    enum socket is {LGA1156, ASB1};
3  }
```

For enumerations, the **in** is mandatory. Notice the use of curly braces here as opposed to square brackets for the *price* attribute above. In TVL, square brackets are used to declare intervals and braces to declare lists. Enumeration are very similar to *xor*-decomposed

features: they can be seen as a shorthand notation which avoids clutter and boilerplate code.

In many cases, the value of an attribute will be calculated based on the values of some other attributes. The value of the *price* attribute of *Accessories*, for example, is the sum of the prices of its children *KeyboardAndMouse*, *PhilipsScreen* and *SamsungScreen*. Furthermore, the value of an attribute might also depend on whether its containing feature is selected or not. All this is written as follow in TVL:

```
1  Accessories {
2    int price is sum(selectedChildren.price)
         ;
3    group [0..2] {
4      KeyboardAndMouse{
5        int price is 19;
6      },
7      PhilipsScreen {
8        int price is 99
9      },
10     SamsungScreen {
11       int price, ifIn: is 149, ifOut: is
             0;
12     }
13   }
14 }
```

The keyword **is** can be used to set the value of an attribute, e.g. *Accessories*, *KeyboardAndMouse* and *SamsungScreen*. The keywords **ifIn:** and **ifOut:** are guards that allows to specified the value of the attribute in the case in which the containing feature is selected (**ifIn:**) or not selected (**ifOut:**). They illustrate this with the *price* attribute of the *SamsungScreen* whose value will be 149 if the feature is selected and 0 if not.

While the price of the *KeyboardAndMouse*, *PhilipsScreen* and *SamsungScreen* features is fixed, the price of the *Accessories* is calculated: it is the sum (using the aggregation function **sum**) of the values of the *price* attribute of its selected children (using the **selectedChildren** keyword, which basically represents the list of values of an attribute declared in all the selected child features). Other operators are available and will be discussed in next paragraph. A common modelling pattern for attributes declared for all feature is to compute the value of the parent feature's attribute by aggregating the attribute values of its children, up to the root. The price of a *Computer*, for example, will be calculated by summing the prices of its selected sub-features, which in turn depend on the prices of their sub-features, and so on until leaf features with fixed price values are reached.

Expressions

In TVL, expressions are used to determine the value of an attribute (as explained in the previous paragraph) as well as to express constraints on the FM (detailed in the following paragraph). The language is strongly typed, each expression being either of type *bool*, *int* or *real*. For more info about types, see [12, Section 4].

A basic expression is either an integer, a real, a Boolean, or a reference to a feature, an attribute or a constant. Those basic expressions can then be combined using classical operators: +, -, /, *, **abs** for numeric values; **!**, **&&**, ||, ->, <-> for Boolean values as well as comparison operators >, >=, < or <=. Classical FM cross-tree constraints **excludes** and **requires** can also be used as Boolean expressions.

Furthermore, there are a number of aggregation functions **sum**, **mul** (multiplication), **min**, **max**, **avg** (average), **count**, **and**, **or** and **xor**. These aggregation functions can simply be used on lists of expressions or they can become powerful shorthand notations

when used in combination with the **children** or the **selectedChildren** keywords. These allow to aggregate the value of an attribute that is declared for each child of a feature. The notation is **fct(children.attribute)**, or **fct(selectedChildren.attribute)** if the aggregation should be calculated on selected children only.

A full listing of the expressions syntax is given in [12, Section 3.5].

Constraints

Constraints in TVL are attached to features (classical constraints that hold 'for the whole model' can be attached to the root, for instance). They are simply Boolean expressions that can be added to the body of a feature definition, as with attribute declarations. They are terminated by a semicolon. The **ifIn:** and **ifOut:** guards we have previously seen can be used on constraints, too. In the computer example, the *Motherboard* feature has a *socket* attribute. The value of this attribute depends on the choice of the actual motherboard, i.e. on the choice of one of the sub-features. One way to model this in TVL is to define a constraint in each child feature which basically 'sets' the value of its parent's attribute.

```
1  Motherboard {
2    enum socket in {LGA1156, ASB1};
3    group oneOf{
4      Asus{
5        ifIn: parent.socket == LGA1156;
6      },
7      Aopen{
8        ifIn: parent.socket == ASB1;
9      }
10   }
11 }
```

The constraint is guarded by **ifIn:**, which means that it is only applicable if the containing feature is selected.

Data Blocks

TVL can serve as an extensible storage format for feature modelling tools. It is possible to attach to every feature a catalogue of *key/value* pairs which contain additional, tool-specifc, data. If, for instance, TVL were to be used as the storage format for a graphical FM tool, the data block of a feature might contain the coordinates of the feature on the screen and other style information:

```
1  Computer {
2    data{
3      "xPos" "123";
4      "yPos" "456";
5    }
6  }
```

Data blocks are the only part of the language that does not have a meaning in terms of FMs. Their contents cannot be 'referenced' anywhere in the model.

Modularisation mechanism

One of the main goals for, the authors, in designing TVL is modularity (to achieve scalability). TVL thus offers various mechanism that can help users to modularise large models. First of all, custom types can be defined on at the top of the file and then be used in the FM. This allows to factor out recurring types and can thus reduce consistency

errors. For instance, one might want to define the different sockets upfront and then use it as a type in an attribute declaration:

```
1 enum cpuSocket {LGA1156, ASB1};
2 ...
3 Motherboard {
4   cpuSocket socket;
5 }
```

It is possible to define structures types to group attributes that are logically linked. A *dimension*, for instance, is a couple (height, width) and can be declared as such using a structured type. This type can then be reused inside the *Motherboard* feature:

```
1 struct dimension {
2   int height;
3   int width;
4 }
5 ...
6 Motherboard {
7   dimension size;
8 }
```

Users can also specify constants using the **const** keyword followed by a type, a name and a value. These constants can then be used inside expressions or cardinalities.

```
1 const int maxRamBlock 4;
```

Modularisation is achieved through two distinct mechanisms. The first is the **include** statement, which takes as parameter the path of a file (relative to the file containing the root feature). As expected, an **include** statement will include the contents of the referenced file at this point. Includes are in fact preprocessing directives and do not have any meaning beyond the fact that they are replaced by the referenced file.

```
1 include(./some/other/file);
```

The second mechanism is that features can be defined at one place and then be extended further in the code. This has two consequences: the definition of a feature can be spread over number of blocks and the physical hierarchy of the FM does not have to be maintained inside the code (for instance, to break up deeply nested hierarchies requiring lots of indentations).

Basically, once a feature has been defined in the group block of its parent feature, its definition can be extended any number of times. In order to extend a feature definition, one just adds a feature block with the same name to the file. This block cannot be inside another feature, it has to start its own hierarchy. Each feature block may add constraints and attributes to the feature body. The children (with the **group** keyword) can only be defined in a single one of these blocks.

This mechanism allows modellers to organise the FM according to their preferences and can be used to implement separation of concerns [36]. For example, one could separate different attribute concern (e.g. attributes related to price and attributes related to technical detailed, like the sockets). Another scenario would be to declare part of the structure of the FM without detailing each feature's attributes and instead provide them later on:

```
1 root Computer {
2   group all Of {
3     Motherboard,
```

```
4       CPU,
5       GraphicCard,
6       opt Accessories
7     }
8   }
9   Computer {
10    int price is sum(selectedChildren.price)
          ;
11  }
12  ...
13  Motherboard{
14    dimension size;
15    ...
16  }
17  ...
```

In this example, the decomposition of the top feature *computer* is defined at the beginning while its attributes and those of *Motherboard* are declared further down. The advantage of this is that the structure is easily understandable because it is not cluttered by attributes of different features.

3.3.2 Benefits & Disadvantages[10]

TVL aims to offer a solution with the main benefit of FM that could be really used in the industry, but it doesn't pretend to replace FD. And TVLs offers some benefits, like the following:

B1: **Providing modularity mechanisms** such that presented in the previous subsection, TVL allows to organize the structure of models and separate the concerns. For instance, thanks to the inclusion mechanism, features belonging to a specific problem can be isolated in a dedicated file. Thus, each problem can be more easily manage while the focus is on local pertinent feature.

B2: **Plain text notation** has a number of advantages the most important of which is the abundance of established tools dealing with text, generally program code. Moreover, TVL is inspired by the syntax of C and should appear intuitive to any engineer who has come in contact with one of the many programming languages with C-like syntax. So TVL does not require dedicated modelling tools to be deployed. Furthermore, TVL is concise, its syntax is very light, as opposed to XML, for instance.

B3: **Definition through a formal semantics** makes the comparison with other languages easier. Thus, syntactic equivalences between TVL and another language (that have also a well-defined semantics) can be define. In this case, for instance, TVL can be used as an exchange format, to export FM from software with a private language (syntax and semantics are unknown by public, usual in industry). TVL can also be used as a storage format for graphical software's. Thanks to data blocks, this tools can store their own informations inside models. Furthermore, the formal semantics makes TVL models automatically verifiable with dedicated tools. Those tools could check constraints validity (e.g. no conflict between constraints), detect dead features (e.g. features that can not be in any configuration[11]), etc.

B4: **Learning experience** for software engineers with a good knowledge of programming languages, especially languages with a C-like syntax, ends with a gentle learning

[10] Based on the TVL case studies [20] and the master's thesis of P. Faber[17]
[11] See definition at Section 4.2

curve. Software engineers who use only graphical models might need a little more time to feel comfortable with the syntax.

However according to some case studies, TVLs required more effort to be implemented and some factors like the following can compromise it:

D1: **Verification of the scope of features** in constraints is one of the requested functionalities. Since a constraint can contain any feature in the FM, it might rapidly become hard to identify whether the referenced feature is unique or if a relative path has to be given. The on-the-fly suggestion of alternative features by the editor would facilitate constraint definition and make it less error-prone. By extension, the on-the-fly checking of the satisfiability of the model would avoid wasting time later on debugging it. The downside of such checks is that they can be resource-intensive and should thus be carefully scheduled and optimized.

D2: **Missing constructs** have been pointed out in TVL. First, default values which are useful to speed up product configuration by pre-defining values. Secondly, feature cloning is missing and will not be introduced in TVL until all the reasoning issues implied by cloning is solved (this is work in progress). Thirdly, attribute cardinality should be available and is technically a simple extension of the decomposition operators defined for features.

D3: **Specification of error, warning and information messages** are needed input for a configurator and so should be directly within the TVL model. These messages are not simple comments attached to features but rather have to be considered as guidance provided to the user that is based on the current state of the configuration.

Chapter 4

Configuration systems

FMs are generally used to represent a family of related programs, like an SPL. When the units of program construction are features, every program in an SPL is identified by an unique and legal combination of features aka. a *configuration*, and vice versa. FMs clarify the constraints between different features provided by a system. Constraints can be represented as a relationship between a parent feature and its children (sub-features) or expressed with additional cross-tree logical constraints. [25, 27]

Information can be automatically extracted from FMs using automated mechanism, called *automated analysis*. Analysing such models is an error-prone and tedious task, and it is infeasible to do manually with large-scale FMs. Currently, a number of operations of analysis, tool, paradigms and algorithms to support the analysis process [6]. The most popular analyses and some tool are exposed in this chapter.

4.1 Analysis operations on Feature Models[1]

This section revisits some of the most popular analyses on FMs and reasoning techniques. A more comprehensive list is available in the systematic review of Benavides et al. [6].

4.1.1 Analyses

For each operation, its definition, an example and possible applications are presented.

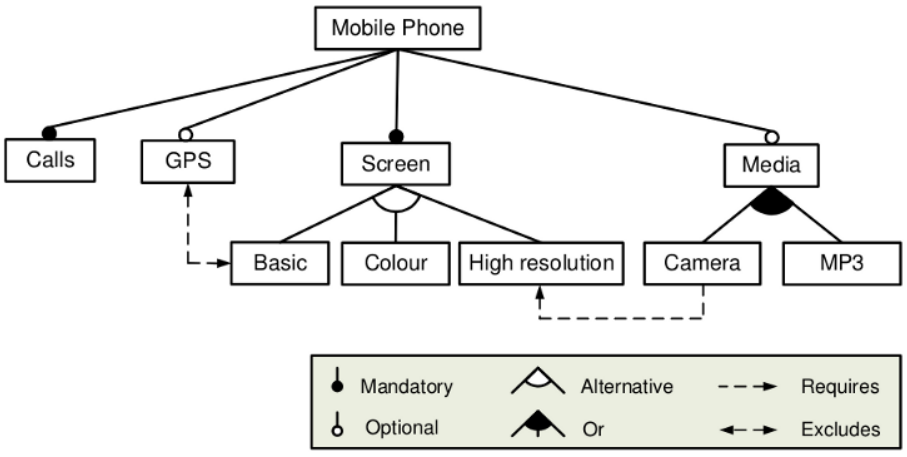


Figure 4.1.1: A sample feature model [6]

[1]Based on A. Hubaux's PhD thesis [19, section 2.4] and the systematic review of Benavides et al. [6].

Satifiability checking. An FM is *satisfiable* when at least one configuration, also called product, can be derived from it. An unsatisfiable FM is synonym of an over-constrained model from which no product can be derived, i.e. FMs without cross-tree constraint are all satisfiable. As an example, Figure 4.1.2 depicts an unsatisfiable FM. Constraint *C-1* (excludes) makes the selection of the mandatory features B and C not possible, adding a contradiction to the model because both feature are mandatory.

The automation of this operation is especially helpful when debugging large scale FMs in which the manual detection of errors is recognized to be an error-prone and time-consuming task [6].

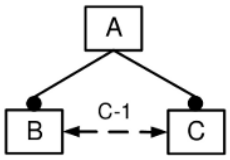


Figure 4.1.2: Unsatisfiable FM [6]

Configuration validation. A configuration is *legal* or *valid* if it satisfies the FM. It is equivalent to a product that belongs to the set of product represented by the feature model. For instance, consider the products[2] $P1$ and $P2$, described below, and the FM of Figure 4.1.1.

$$P1 = \{MobilePhone, Screen, Colour, Media, MP3\}$$
$$P2 = \{MobilePhone, Calls, Screen, Highresolution, GPS\}$$

Product $P1$ is not valid since it does not include the mandatory feature *Calls*. On the other hand, product $P2$ does belong to the set of products represented by the model.

This operation may be helpful for SPL analysts and managers to determine whether a given product is available in an SPL [6].

Partial configuration validation. By extension, a *partial* configuration is legal if it is a subset of a legal configuration, i.e. it does not include any contradiction. Consider as an example the partial configurations[3] $C1$ and $C2$, described below, and the FM of Figure 4.1.1.

$$C1 = (\{MobilePhone, Calls, Camera\}, \{GPS, Highresolution\})$$
$$C2 = (\{MobilePhone, Calls, Camera\}, \{GPS\})$$

$C1$ is not a valid partial configuration since it selects support for the camera and removes the high resolution screen that is explicitly required by the SPL. $C2$ does not include any contradiction and therefore could still be extended to a valid full configuration.

This operation results helpful during the product derivation stage to give the user an idea about the progress of the configuration. A tool implementing this operation could inform the user as soon as configuration becomes invalid, thus saving time and effort [6].

[2]Set of features to be selected.

[3]2-tuple of the form (S, R) where S is the set of feature to be selected and R the set of features to be removed such that $S \cap R = \emptyset$.

Dead feature detection. A feature that can not appear in any configuration is called *dead*. Dead features are caused by a wrong usage of cross-tree constraints. These are clearly undesired since they give the user a wrong idea of the domain. Figure 4.1.3 depicts some typical situations that generate dead features [6].

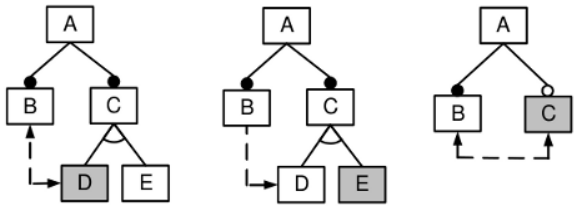


Figure 4.1.3: Common cases of dead features (grey features are dead) [6]

Core features detection. *Core features* are features that appear in every configurations. For instance, the set of core feature of the model presented in Figure 4.1.1 is $\{MobilePhone, Calls, Screen\}$.

Core features are the most relevant features of the SPL since they are supposed to appear in all products. Hence, this operation is useful to determine which features should be part of the core architecture of the SPL [6].

Atomic set computation. An *atomic set* is a group of features (at least one) that can be treated as a unit when performing certain analyses. The intuitive idea behind atomic sets is that mandatory features and their parent features always appear together in products and therefore can be grouped without altering the result of certain operations. Once atomic sets are computed, these can be used to create a reduced version of the model simply by replacing each feature with the atomic set that contains it. Figure 4.1.4 depicts an example of atomic sets computation. Four atomic sets are derived from the original model, reducing the number of features from 7 to 4. Note that the reduced model is equivalent to the original one since both represent the same set of products.

Using this technique, mandatory features are safely removed from the model. This operation is used as an efficient preprocessing technique to reduce the size of feature models prior to their analysis [6].

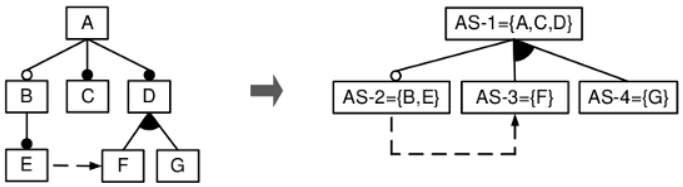


Figure 4.1.4: Atomic sets computation [6]

Explanations delivery. This operation delivers informations about *why* or *why not* the corresponding response of the operation. Causes are mainly described in terms of features and/or relationships involved in the operation and explanations are often related to anomalies. For instance, Figure 4.1.5 presents a FM with a dead feature. A possible explanation for the problem would be *"Feature D is dead because of the excludes constraitn with feature B."*

Explanations are a challenging operation in the context of FM error analysis. In order to provide an efficient tool support, explanations must be as accurate as possible

when detecting the source of an error, i.e. it should be minimal. This becomes an even more challenging task when considering extended FMs and relationships between feature attributes [6].

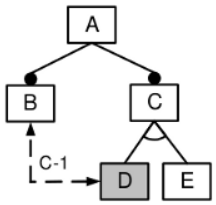


Figure 4.1.5: Grey feature is dead because relationship C-1 [6]

4.1.2 Automated supports

These operations can be automatically verified with appropriate solvers and formal encoding. Benavides et al. [6] identify four groups of reasoning approaches according to the logic paradigm or method used to provide the automated support.

Propositional logic based analyses. A *propositional formula* consists of a set of Boolean variables and a set of logical connectives constraining the values of the variables, e.g. $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\Leftrightarrow$.

A *basic feature model*[4] and primary cross-tree constraints (e.g. implies and excludes) can be expressed in Propositional Logic (PL), as shown in Figure 4.1.6.

Two types of solvers are commonly used to efficiently handle FMs encoded as propositional formulae. *Boolean Satisfibility Problem (SAT) solvers* (e.g. SAT4J [8] take as input a propositional formula and determines of the formula is satisfiable, i.e. there is a variable assignment that makes the formula evaluate to true. SAT solvers usually require the propositional formula to be converted in a Conjunctive Normal Form (CNF). A CNF is a conjunction of clauses where each clause is a disjunction of variables such that a variable and its complement cannot appear in the same clause. It has been proved that every propositional formula can be converted into an equivalent formula in CNF formula. This transformation is based on rules about logical equivalences: the double negative law, De Morgan's laws, and the distributive law. CNF formulas are generally recorded in the DIMACS CNF format[11] that is widely accepted as the standard format. The DIMACS CNF format is an ASCII file format. The file starts with comments (each line starts with c) then come the number of variables and the number of clauses defined by a line of the form p cnf variables clauses. Each of the next lines specifies a clause: a positive literal is denoted by the corresponding number, and a negative literal is denoted by the corresponding negative number. The last number in a line should be zero [1]. For example, the following sample DIMACS file represents the formula $(A \vee \neg C) \wedge (B \vee C \vee \neg A)$:

```
1  c A sample .cnf file.
```

[4] A basic feature model is a Boolean feature model where the only relationships between a parent feature and its child features (or sub-features) are categorized as:

- Mandatory - child feature is required.
- Optional - child feature is optional.
- Or - at least one of the sub-features must be selected.
- Alternative (xor) - exactly one of the sub-features must be selected

```
2  p cnf 3 2
3  1 -3 0
4  2 3 -1 0
```

Listing 4.1: Sample DIMACS file

Similarly, a Binary Decision Diagram (BDD) solver (e.g. JavaBDD [41]) takes a propositional formula as input (not necessarily in CNF) and translate it into a DAG (the BDD itself) with two terminal nodes respectively representing `true` and `false`. Each decision node is labelled with a Boolean variable, and has exactly two output edges respectively capturing the assignment of `true` or `false` to the variable. This graph representation allows determining if the formula is satisfiable and providing efficient algorithms for counting the number of possible solutions.

SAT and BDD solvers differ in that SAT solvers suffer from high complexity in time whereas BDD solver suffer from high complexity in space. In practice, both solvers complement each other and should be picked carefully based on the type of analysis to conduct [27].

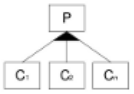
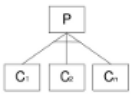
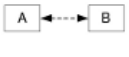
Relationship		PL Mapping	Mobile Phone Example
MANDATORY		$P \leftrightarrow C$	MobilePhone $\leftrightarrow$ Calls MobilePhone $\leftrightarrow$ Screen
OPTIONAL		$C \rightarrow P$	GPS $\rightarrow$ MobilePhone Media $\rightarrow$ MobilePhone
OR		$P \leftrightarrow (C_1 \vee C_2 \vee ... \vee C_n)$	Media $\leftrightarrow$ (Camera $\vee$ MP3)
ALTERNATIVE		$(C_1 \leftrightarrow (\neg C_2 \wedge ... \wedge \neg C_n \wedge P)) \wedge$ $(C_2 \leftrightarrow (\neg C_1 \wedge ... \wedge \neg C_n \wedge P)) \wedge$ $(C_n \leftrightarrow (\neg C_1 \wedge \neg C_2 \wedge ... \wedge \neg C_{n-1} \wedge P))$	(Basic $\leftrightarrow$ ($\neg$Color $\wedge$ $\neg$Highresolution $\wedge$ Screen))$\wedge$ (Color $\leftrightarrow$ ($\neg$Basic $\wedge$ $\neg$Highresolution $\wedge$ Screen))$\wedge$ (Highresolution $\leftrightarrow$ ($\neg$Basic $\wedge$ $\neg$Color $\wedge$ Screen))
IMPLIES		$A \rightarrow B$	Camera $\rightarrow$ Highresolution
EXCLUDES		$\neg(A \wedge B)$	$\neg$(GPS $\wedge$ Basic)

Figure 4.1.6: Mapping from feature model to propositional logic (adapted from [6])

Constraint programming based analyses. A Constraint Satisfiability Problem (CSP) is a mathematical problem defined as a set of objects that must satisfy a number of constraints or limitations. *Constraint programming* can be defined as the set of techniques such as algorithms or heuristics that deals with CSPs. A CSP is solved by finding states (values for variables) in which all constraints are satisfied. The mapping from an FM to a particular CSP solver is less straightforward than with propositional logic because each solver has its own encoding scheme. Moreover, CSP often have high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable

time. However, in contrast to propositional formulas, CSP solvers can deal with numerical values such as integers or intervals, not only with binary values (true or false) [6].

Description logic-based. Description Logic (DL) defines a family of formal languages meant to conceptualise, and reason about knowledge. Essentially DL allow to model concepts, roles (properties on concepts and relationships among them), and individuals (instances). A DL reasoner takes as input a problem described in DL and provides facilities for consistency and correctness checking and other reasoning operations [6].

Other. Other approaches rely on original algorithm especially produce for FMs. These algorithms are usually bound to a particular dialect of FMs and focus an very specific analyses. Further information is available in the Benavides et al.'s article [6].

> Solvers not only help improve the quality and correction of FMs. They also provide the backbone that maintains the consistency of the configuration throughout the application engineering process. [19]

4.2 Feature-based Configuration[5]

Feature-based Configuration (FBC) is the interactive process during which the selection of features to be included in a product is performed by the stakeholders. FBC systems (aka. *configurator*) have to ensure the validity of the configuration along the whole process. They should be able to manage an interactive setting: 1. automatically *propagating* decisions, and 2. *explaining* the results of the propagation [19].

A. Hubaux defines the *decision propagation* as the procedure that automatically sets the values of variables that depend on the decision. In other words, it automatically selects feature required by the decision and respectively deselects excluded features. Thus, a product always remind satisfiable during the configuration, no decision can break it. As previously introduced, an *explanation* is the feedback delivered to stakeholders that details how a decision was made, i.e., manually or automatically. In the automatic case, the explanation should contain the manual decision that induced the propagation, and the constraints involved by the (de)selection. Note that several explanations can match a single decision [19].

In practice, performance, scalability, and integration requirements also prevail. FMs with thousands of features constrained by quantities of complex constraints show an high level of complexity that require extremely efficient reasoners to propagate decisions while preserving the reactivity of the configuration interface [19].

This section presents some configuration tools split into two different categories:1. Generic; and 2. Domain specific. This presentation aims to provide a glimpse of the configuration management in practice through key examples that will support our later reflection.

4.2.1 Generic Feature Model Tools

Over the years, the formalisation of FMs helps to support FBC and develop interactive configurators. Propagation of decisions throughout the FM inside configurators relies on efficient solvers (typically SAT, BDD and CSP solvers) that excel in the domain. As said in Section 4.1.2, their reasoning abilities depend on the type of the solver. For instance, Software Product Lines Online Tool (SPLOT) [28] uses SAT and BDD only, which

[5]Based on A. Hubaux's PhD thesis [19, section 2.5]

restricts the reasoning domain to binary decision but enables extremely fast reasoning about thousands of features [19]. This tool is further detailed below. On the other hand, a tool using a dialect like Prolog (e.g., pure::variants [34]) gains in expressiveness, thanks to the general-purpose logic programming language and its support of arbitrary domains like integers or strings. But in return, it has lower performances. Usually, these tools also propose other analyses such as explanation, dead feature detection, core feature detection, model comparison, model metrics computation, and auto-completion [19]. Other tools like FeatureIDE [38], focus on *feature-oriented programming*, and create a bridge between an FM and a code base. The configuration of the FM allows to remove unnecessary source code from a particular product [19]. FeatureIDE is further detailed below.

All these tools are not domain dependant and could be characterized as *generic feature model tools*.

Software Product Lines Online Tool[6]

SPLOT is a Web-based reasoning and configuration system for SPLs. SPLs are represented by FMs with additional cross-tree constraints in CNF. SPLOT, developed in Java[7], is supported by sophisticated configuration engines and efficient automated reasoning systems based on public BDD and SAT engines, respectively JavaBDD and SAT4J [8]. However, it is well-known that BDDs and SAT solvers can suffer from space (BDD) and/or time (SAT) intractability problems. To minimize such problems, SPLOT make use of novel BDD heuristics [27] to reduce the size of BDDs as much as possible. As well, the system capitalizes on the observed efficiency of SAT systems in the feature modelling domain [29] to provide high-performance SAT-based algorithm (e.g. valid domain computation) [28].

Specifically, SPLOT uses a BDD engine to count valid configurations, to calculate the variability degree of feature models and to perform interactive configuration. Moreover, a SAT solver is also used to support interactive configuration and to perform debugging tasks such as checking consistency of FMs and detecting common and dead features [28].

Currently, SPLOT provides two major services: *automated reasoning* and *product configuration*. Reasoning focus on automating statistics computation and debugging tasks. With regards to product configuration, SPLOT supports *interactive configuration* in which users make a decision at a time and the configuration system automatically propagates those decisions to enforce their consistency [28].

Figure 4.2.1 represents the product configuration with SPLOT. The FM is displayed as a collapsible hierarchical tree and user can interactively select or deselect a feature. User's decision are automatically propagated and each step is stored in a steps-history (on the right). To automatically complete the configuration, users can used the auto-completion with less or more feature which respectively means to attempt to deselect/select all remaining features.

FeatureIDE[8]

FeatureIDE is an Eclipse-based IDE that supports all phases of feature-oriented software development for the development of SPLs: domain analysis, domain implementation, requirements analysis, and software generation. Different SPL implementation techniques are integrated such as feature-oriented programming (FOP), aspect-oriented programming (AOP), delta-oriented programming (DOP), and preprocessors. Currently, FeatureIDE

[6] This content summarizes the SPLOT's official reference [28]

[7] Java Servlet API 2.5

[8] This content summarized pieces of informations available at the FeatureIDE website [38]

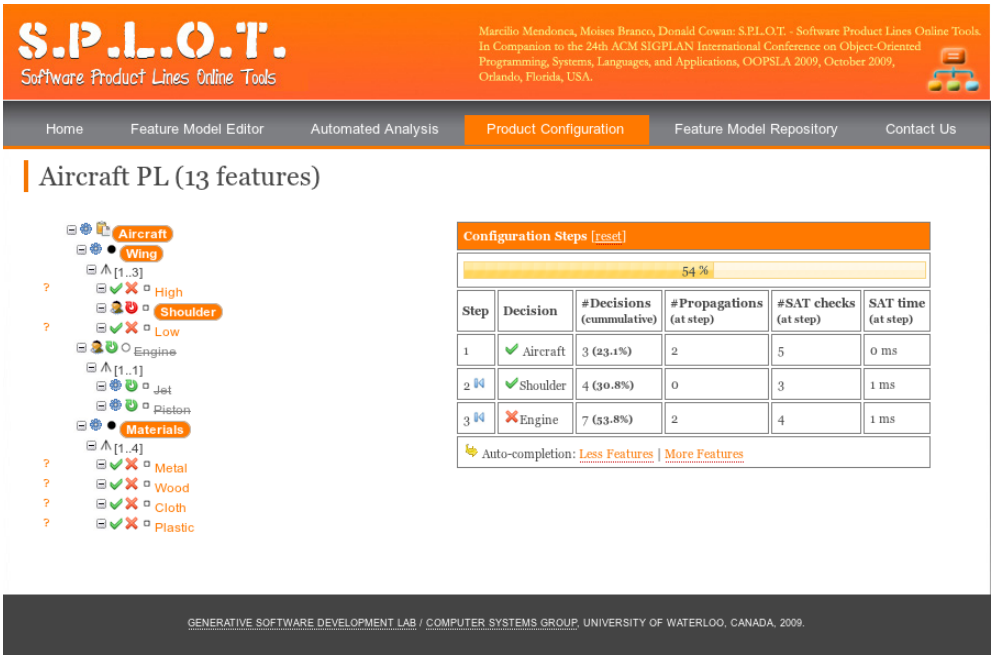


Figure 4.2.1: SPLOT: Product configuration

provides tool support for AHEAD, FeatureC++, FeatureHouse, AspectJ, DeltaJ, Munge, and Antenna [38].

FeatureIDE is under constant development. Currently, it's fully integrated into Eclipse and provides many features like a *FM editor* (graphical[9] and XML based), a *constraint editor* with content assist, *syntax, and semantic checking* (e.g. dead feature detection) and a *configuration editor* to create and edit configurations (Figure 4.2.2b). The configuration editor comes with support for deriving valid configuration [38].

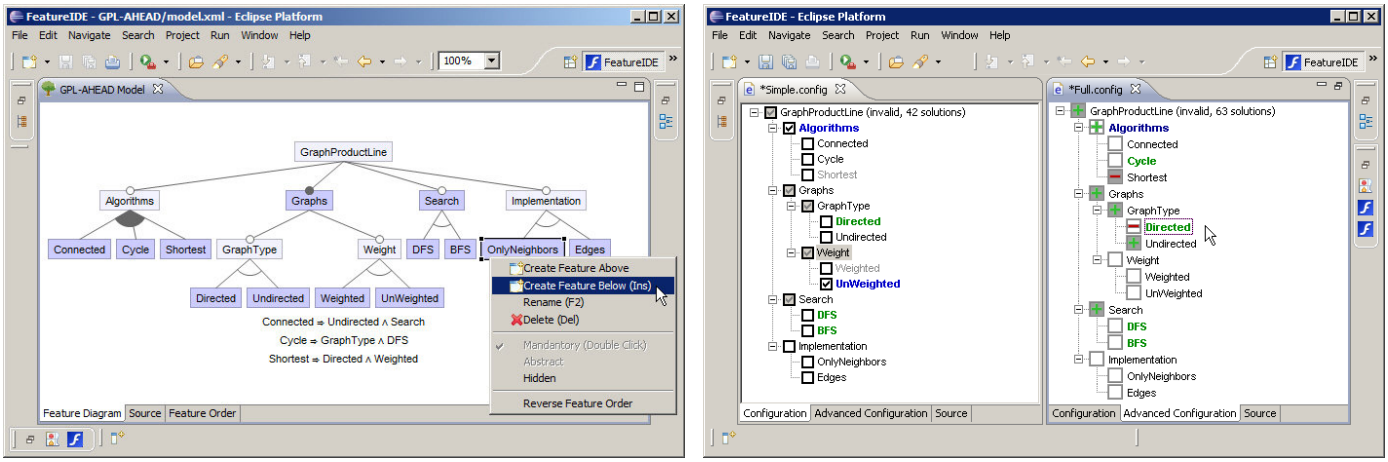
Figure 4.2.2a depicts the graphical edition of a FM with FeatureIDE. FeatureIDE supports *and*-, *or*- and *xor*-decompositions, mandatory and optional features, plus abstract features i.e. features only used to structure the model and selecting or eliminating them does not make any difference in the generated variant code (Figure 4.2.3) [37]. Under the FD, three cross-tree constraints are displayed in propositional logic.

Figure 4.2.2b depicts the configuration edition with FeatureIDE. Two views are available, the main difference is that users can specified undesired features in the 'Advanced configuration' (red minus). In both, user's decisions propagations are displayed in grey. Next to the root feature name, between brackets, the validity of the configuration and the number of solution according to the current configuration are displayed. So, users can easily create a valid configuration.

4.2.2 Domain-specific configurators

Other configurator primarily designed for a specific domain are also available like LKC detailed below. These tools emerge from requirement appearing through the domain evolution. They are conceived to accomplish a dedicated task with the same modelling and reasoning foundations as generic tools.

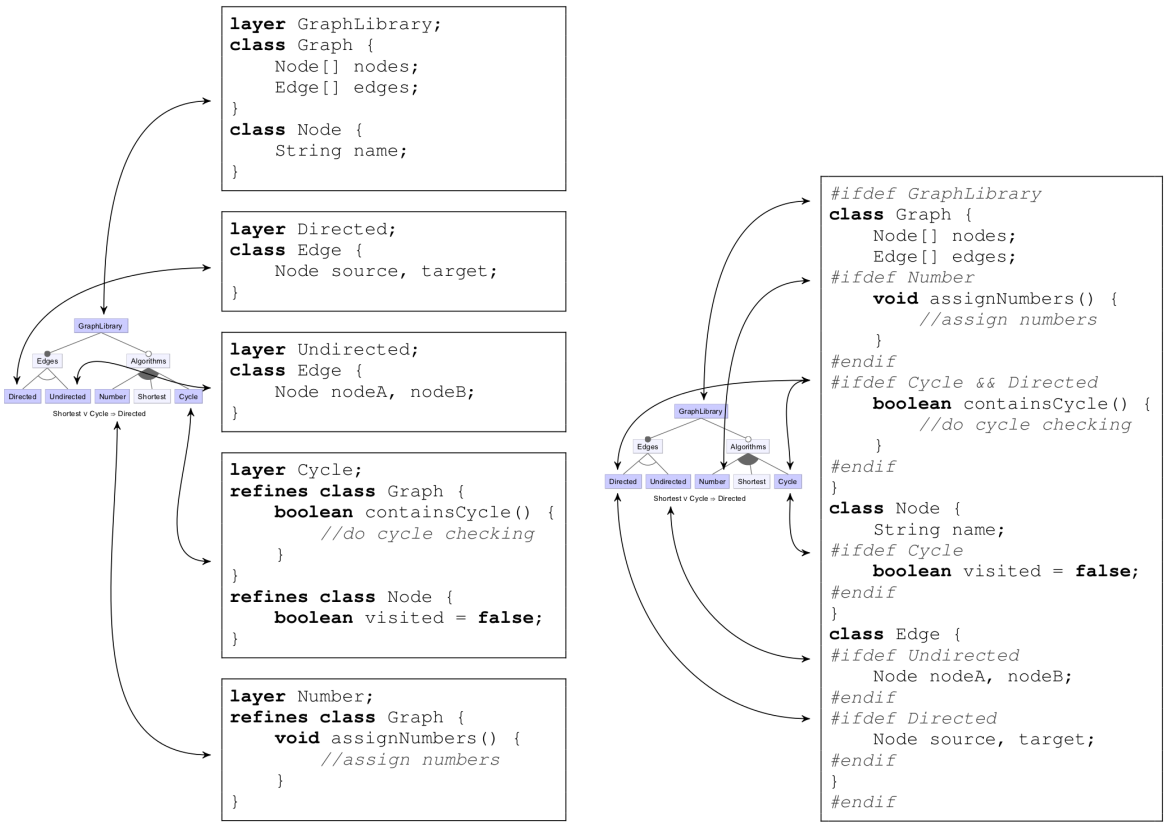
[9] Figure 4.2.2a



(a) Graphical FM editor [38]

(b) Configuration editor [38]

Figure 4.2.2: FeatureIDE screenshots



```
layer GraphLibrary;
class Graph {
    Node[] nodes;
    Edge[] edges;
}
class Node {
    String name;
}
```

```
layer Directed;
class Edge {
    Node source, target;
}
```

```
layer Undirected;
class Edge {
    Node nodeA, nodeB;
}
```

```
layer Cycle;
refines class Graph {
    boolean containsCycle() {
        //do cycle checking
    }
}
refines class Node {
    boolean visited = false;
}
```

```
layer Number;
refines class Graph {
    void assignNumbers() {
        //assign numbers
    }
}
```

(a) A 1:1 mapping between features and feature modules using feature-oriented programming [37]

```
#ifdef GraphLibrary
class Graph {
    Node[] nodes;
    Edge[] edges;
#ifdef Number
    void assignNumbers() {
        //assign numbers
    }
#endif
#ifdef Cycle && Directed
    boolean containsCycle() {
        //do cycle checking
    }
#endif
}
class Node {
    String name;
#ifdef Cycle
    boolean visited = false;
#endif
}
class Edge {
#ifdef Undirected
    Node nodeA, nodeB;
#endif
#ifdef Directed
    Node source, target;
#endif
}
#endif
```

(b) An n:m mapping between features and preprocessor statements [37]

Figure 4.2.3: FeatureIDE mappings for code generation

Linux Kernel Configurator[10]

LKC is a tool that is delivered within the Linux Kernel in order to enable its configuration (feature selection). Its first prototype was proposed in 2002, the current version is 1.4, and as the Linux Kernel, it is released under the GNU General Public License [35].

In 2001, the community around the Linux Kernel started to show dissatisfaction with the kernel configuration tool, back then known as configuration menu language (CML1). With the growth of the kernel, the configuration process was getting very complicated. The tool was responsible for selecting the capabilities to be built into the kernel, handling dependencies and providing the user interface for feature selection. Moreover, it was comprised of a mixture of code written in Tcl/Tk scripts, awk scripts, pearl and C, which made it hard to understand and to maintain. In order to solve this problem, the LKC was finally proposed [35].

LKC is basically comprised of a *parser* and a *dependency checker* that are used as the back-end. To enable the selection of *configuration options* aka. features (as defined in a configuration database aka. FM), different front-ends (graphical, text-mode, command-line interactive, etc.) are provided [35].

The configuration file is a text file containing the entries which must follow a strict syntax. The FM is built as a set of *entries* which define features. Such a file is depicted in the example (Listing 4.2 p.39) provided by Julio Sincero and Wolfgang Schröder-preikschat [35].

> Line 1 of Listing 4.2 p.39 shows an entry definition, it starts with the keyword **config** and is followed by its name. The next lines of an entry are used to define its attributes, which can be the following:
>
> **type** define the type of an entry: `boolean`, `tristate`[11], `string`, `hex` and `integer`.
>
> **input prompt** is the visual name of the feature that is displayed to the user during configuration. On line 1 the actual configuration name is defined as `GPL` which will be used in the generated configuration file, however, the user will see during configuration the name `ROOT` as shown on line 2.
>
> **default value** is assigned to the configuration symbol if no value was set by the user. An example is given on line 17.
>
> **dependencies** define the requirements of the menu entry. They can simply define the depending on a single feature, as shown on line 6, or can be in the form of logical expression using primitives like && (logical and), || (logical or), as shown on line 18.
>
> **reverse dependencies** are used to forced the lower limit of the value of another symbol. As shown on line 3, if the symbol GPL is selected, the symbol M1 will automatically be selected as well.
>
> **numerical ranges** limit the range of possible input values for `integer` and `hex` symbols.
>
> **help text** defines the feature help text to be shown during configuration. Examples are shown on line 21 and 31.

[10] This content summarizes the paper of Julio Sincero and Wolfgang Schröder-preikschat [35]

[11] The boolean type can be assigned to yes or no, the tristate type allows an extra value (m) which means that the configuration option should be included, however, as a separate module.

Moreover, in order to provide a better organization of the entries in the configuration tree that is displayed to the user (Figure 4.2.4), the following constructs are allowed:

menu entries defined between the keywords **menu** and **endmenu** are grouped together and displayed in a separated window. It may have an attribute **prompt** to name the groups of entries. An example is given between lines 5 and 14.

choice Only one entry of those defined between the keyword **choice** and **endchoice** can be selected if its parent entry is also selected. [35]

```
1  config GPL
2    boolean "ROOT"
3    select M1
4
5  choice
6    depends on GPL
7    prompt "Graph Type"
8
9    config DIRECTED
10     boolean " Directed"
11
12   config UNDIRECTED
13   boolean "Undirected"
14 endchoice
15
16 config NUMBER
17   default y if GPL
18   requires (BFS || DFS)
19   boolean "Number"
20   ---help---
21   Assigns a unique number to each
22   verte x as a result of a graph
23   traversal.
24
25 config CC
26   depends on GPL
27   requires (BFS || DFS)
28   requires UNDIRECTED
29   boolean "Connected Comp."
30   ---help---
31   Computes the connected components
32   of an undirected graph, which are ...
```

Listing 4.2: LKC language

Figure 4.2.4 depicts the screenshot of the LKC graphical front-end displaying the FM of the Graphical Product Line (GPL). The FM is displayed as a collapsible hierarchical tree. The *xor*-decomposition is represented by a group of radio button and *optional feature* by a check-box [35].

Figure 4.2.5 summarizes the mappings from the LKC language to feature model relations. for *mandatory* relation, the parent feature forces the selection of the child by the use of a reverse dependency (**select**). The *optional* relation is described by using a dependency between the child and the parent feature (**depends on**). The *or group* is designed by creating reverse dependencies between the children and the parent, this is done inside a menu definition in order to group the children together. The *alternative group* can be described by including configuration options (the children) inside a **choice** definition, which has the same semantic as of *alternative group* in FMs [35].

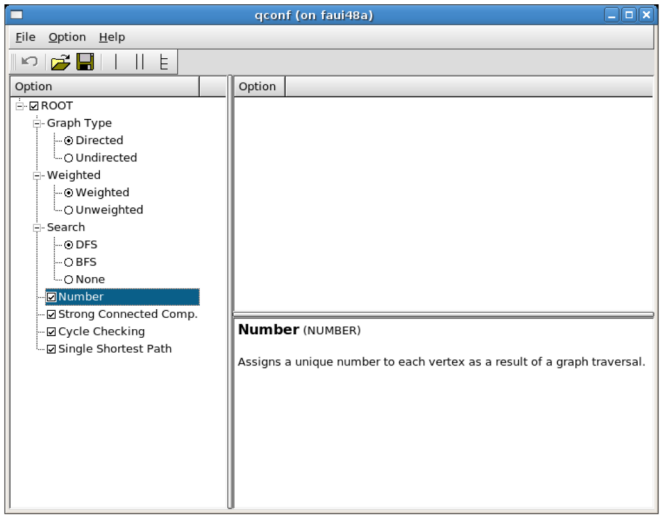


Figure 4.2.4: The LKC graphical interface [35]

4.3 Overview of Configuration beyond Feature Models[12]

This section aims to present, without looking for completeness, the bigger landscape in which FMs fit.

The configuration of physical artefacts, recognition of the business value of customizable software products was the focus of traditional research in the *knowledge-based production configuration* field and led to the emergence of Software Product Line Engineering (SPLE). But even if there is a substantial overlap in research interest of the two fields, they mostly progress on their own path. And some attempts were made aiming to combining the approaches develop in the different fields. Computer-supported configuration of products has a long history, so research on the configuration of parametrisable software is pretty new compared to it [21].

4.3.1 Product configuration[13]

Product Configuration (PC) is defined by A. Hubaux et al. as "the umbrella activity of assembling and customizing physical artefacts (e.g. technical equipment, cars or muesli) or services". It goal is to save cost by assembling individualised systems from reusable components. Historically, PC has been a subfield of Artificial intelligence (AI), focusing on knowledge representation and reasoning techniques to support configuration [21].

A wide range of knowledge modelling approaches (based, e.g., on UML or descriptive logic) and several types of logics and constraints are used by researchers that work on PC. Most results built upon the seminal work on FODA (Section 3.2), but a few SPLE approaches also used UML to capture aspects of configuration knowledge. Models that can be directly translated into a representation processable by a reasoning engine are generally preferred by the AI-rooted PC community. And the formal basis of the most knowledge modelling languages lays the foundation for advanced configuration reasoning techniques (e.g. checking consistency of configurations, completing partial configurations, or supporting interactive configuration processes). In contrast, the SPLE community only started recently a formal foundation of FMs and their analyses [21].

[12] Based on the research roadmap of A. Hubaux et al. [21] and A. Hubaux's PhD thesis [19, Chapter 3]
[13] This sub-section summarizes the content of A. Hubaux et al. [21].

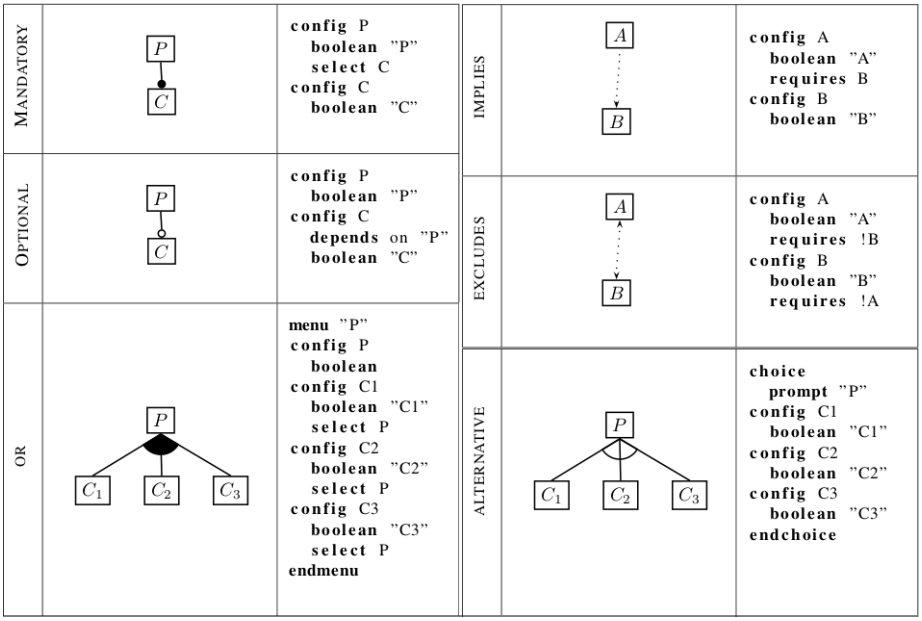


Figure 4.2.5: Mapping: Feature relations to LKC language [35]

In the AI community, it is not new to encode configuration problems in some logic or as CSPs. The PC community was the first to apply SAT solvers to configuration problems. BDDs have also been used for building fast interactive configurators (trading time vs. space from a complexity point of view). Unfortunately, the SPLE community sometimes reinvents technique which have been developed previously in PC [21].

4.3.2 Configuration in manufacturing[14]

Nowaday, a number of tangible products (e.g. cars and mobile phones) as well as intangible products like software (e.g. operating systems and ERPs) and services (e.g. insurance) can be parameterized by customers to fit their preferences. Therefore, software vendors and manufacturing companies have to provide *customisable products* at the same price and conditions than previously standard product or could face to a serious competitive disadvantage [19].

Configurators are commonly used to manage "the sales, product design, and development of manufacturing specifications for customised products" [22]. At the core of the configurator lies the *configuration system*, which is "an expert system that is able to combine modules which are individually described by a number of characteristics, by using rules (constraints) which describe which modules are legal to use in combination" [22].

The decision to create or use a configurator should not be technology-driven but rather result from the clear prevision of commercial advantages like increased customer satisfaction and market share, and reduced production costs. The purpose and design of configurator are also determined by a delivery strategy that prescribes the degree of flexibility and their position in the manufacturing lifecycle [19].

In his thesis A. Hubaux discusses the four main strategies. They are presented below, in crescent complexity order. Some companies combine different strategies to deal with the complexity inherent to the required flexibility of both the manufacturing facilities, the design and specification of the product family [19].

[14]This sub-section summarizes a part of A. Hubaux's PhD thesis [19]

Make-to-stock delivery strategy leads companies to fabricate, assemble and eventually store finished products. In this strategy, the Customer Order Decoupling Point (CODP)[15], also called *order penetration point*, is positioned at the end of the production process. Figure 4.3.1 sketches the position of the CODP for the *make-to-stock* strategy as well as those of three other strategies, which are explained below.

Configure-to-order aka. *assemble-to-order*. In this strategy, products are built based on a combination of standard components pieced together following some choices of the customer like a car.

Make-to-order strategy only starts the manufacturing of already designed components once the order has been placed. So this strategy is more flexible than *configure-to-order*. The fabrication of the pieces is case specific and building more customisable products is easier. For example, hardware manufacturing at Dell is an application of such strategy.

Engineer-to-order strategy leaves a maximum flexibility to the customer. In that case, a substantial amount of work is needed to define accurate specifications. The production of complex plants like cement factories belong to that strategy. The nature and complexity of the products

Product delivery strategy	Design	Fabrication and procurement	Final assembly	Shipment
Make-to-stock	- - - - - -	- - - - - -	- - - - - -→	CODP →
Configure-to-order	- - - - - -	- - - - - -→	CODP ——	——→
Make-to-order	- - - -→	CODP ——	——	——→
Engineer-to-order	CODP ——	——	——	——→

Figure 4.3.1: Different product delivery strategies [19]

determine the type of strategy(ies) that best fits the needs of the company. It is only once agreed upon that the development of the configurator can actually start. [19]

4.3.3 Software Configuration Management[16]

> Configuration management is the art of identifying, organizing, and controlling modifications to the software being built by a programming team. The goal is to maximize productivity by minimizing mistakes. [4]

Software Configuration Management (SCM) introduces the concept of versions that represent instances of products and its parts over time. There are two main types of versions: *revisions* and *variants*. Revisions are versions that replace other version due to bug fixes or addition of new functionalities. Variants are versions intended to coexist through time to satisfy different user or platform needs [21].

SCM gets in every step of the software engineering life-cyle. It also involves many actors with distinct responsibilities, tasks, and expectations from the SCM system. For instance: the *project manager* monitors the progress of the project and makes sure that

[15] CODP is "the point in the manufacturing value chain for a product, where the product is linked to a specific customer order" [33]

[16] This sub-section summarizes a part of A. Hubaux's PhD thesis [19]

the product will be developed within the given time frame; the *configuration manager* enforces development policies (e.g. code creation, change, and testing) and collects data about the status of the project; and the *quality assurance manager* controls the quality of the product. The set of functionalities offered by an SCM system has to cover the needs of every actor [19].

The management of the revisions and the interaction of both is still a weakness of PC and SPLE, especially when the product and its parts evolve frequently. Another practical both in PC and SPLE is the constantly growing number of components that can be part of a configuration [21].

4.3.4 Summary of the overview[17]

According to A. Hubaux, *"the manufacturing industry seems to favour domain-specific interfaces over generic tree-like representations; no matter what reasoning techniques is used in the backend"* [19]. So Graphical User Interfaces (GUIs) should be designed by domain experts to best suit their requirements. And FBC should abstract away from GUIs and focus on its foundation. The FMs is a common representation of the options laid out in particular interfaces. Moreover, A. Hubaux noticed that FBCs lacks integration with other systems or components (e.g. ERP system). Although commercial tools have started integrating other modelling tools, research still a few steps behind. Finally, final users generally prefer easily understandable with no overhead of maintenance. Genericity and enhanced expressiveness may lead to too much flexibility which is often seen as *"a source of confusion, inefficiency, and maintenance overhead"* [19].

Like A. Hubaux, we are aware of the limitations of these conclusions. To provide definitive statements, a more systematic and thorough treatment of the fields is needed but is beyond the scope of this thesis.

[17]This sub-section summarizes a part of A. Hubaux's PhD thesis [19]

Part II

Contribution

Chapter 5

Feature-based configurator

In this chapter, a feature-based configurator prototype, called CAMelot, is presented. At first, we expose our motivation (Section 5.1) and we propose an overview of the suggested system (Section 5.2). Then we explain in details the implementation of CAMelot (Section 5.3): our working assumptions, the system architecture, the main mechanisms, and data generation. Finally, we evaluate our contribution as objectively as possible (Section 5.4).

5.1 Motivation

The primary goal of this thesis is figure out how feature modelling could be helpful in the real case of CAM. As we saw in Section 2.5, CAM has a configuration process designed to support a number of different scientific scenarios. Because of this flexibility, the configuration of CAM is a complex process. To run CAM in the standalone mode, the user must execute a configuration script and provide a number of command line parameters that describe his or her particular configuration choices. Users can only supply parameters that differ from the default values of most configuration options. The final compilation will include only certain source code folders based on the users' choices.

One primary function of the CAM configuration script is to enforce **constraints** among the configuration options. The configurable nature of CAM makes it more like an SPL than a single software application. In other words, the CAM codebase can be used to create a large number of related software applications that vary in different ways. According to this similarity, we'll suggest a re-engineering of CAM configuration script based on feature modelling to **improve maintainability of CAM system** (*Challenge C1*[1]). As FMs offer a convenient and simple notation for documenting common and variation points in SPLs. As well FMs **introduce a separation of concerns** (*Challenge C2*) as A. Hubaux mentions it in his PhD's thesis [19, Chapter 4]. Our work is focussed on the climate model configuration rather than on machine and archiving options but we'll discuss both programmer and user points of views.

Moreover, with their graphical notation, FDs also allow you to describe more or less simple constraints among the features. This should be helpful, at least, for visualizing the variation points (configuration options) of CAM and constraints among the different configuration options [14]. That is a first argument in favor to FMs to **highlight options interactions** (*Challenge C3*). A second argument is the tool support that comes with feature modelling like configurators. These tools offer explanations during the configura-

[1] This challenge is also taken up through the others.

tion process thanks to embedded reasoners (Section 4.2). A feature-based configurator, as the configuration script, would check constraints among the configuration options and would select appropriate source code directories to finally produce a `makefile`[2] and a `filepath`[3] according to the parametrization. But this parametrization must currently be done by a domain expert familiar with the configuration script of CAM, otherwise he or she could receive a series of error messages. A configurator would guide the configuration thanks to decision propagation and users would have **no longer need to be experts** of configuration script to create a configuration (*Challenge C5*).

In addition, this kind of script has a significant risk of becoming inconsistent as systems evolve. The best simple example of inconsistency is conflicting constraints like an exclusive constraint and a required constraint both define on two particular options. **Support the constant evolution** of CAM is a real challenge for the CESM community (*Challenge C6*). Readability and version management should be a starting point to meet this challenge. Generally, FM language don't provide a higher level of readability than a script, but TVL is really light and easy to understand. Version management will not be discussed in this thesis as it's a problem that affects a wider scope than the FM field.

Finally, the **support of default options** (*Challenge C4*) will be discussed in the next chapter as it's not part of FMs and require a language extension. Thus at this stage of our study, we'll omit the support of default values inside our suggested solution.

CAM configuration process may be more explicit with Figure 5.1.1 that shows the current process steps with a Roman numeration and the suggested process steps with an Arabic numeration. This suggested process is described in the next paragraph. The

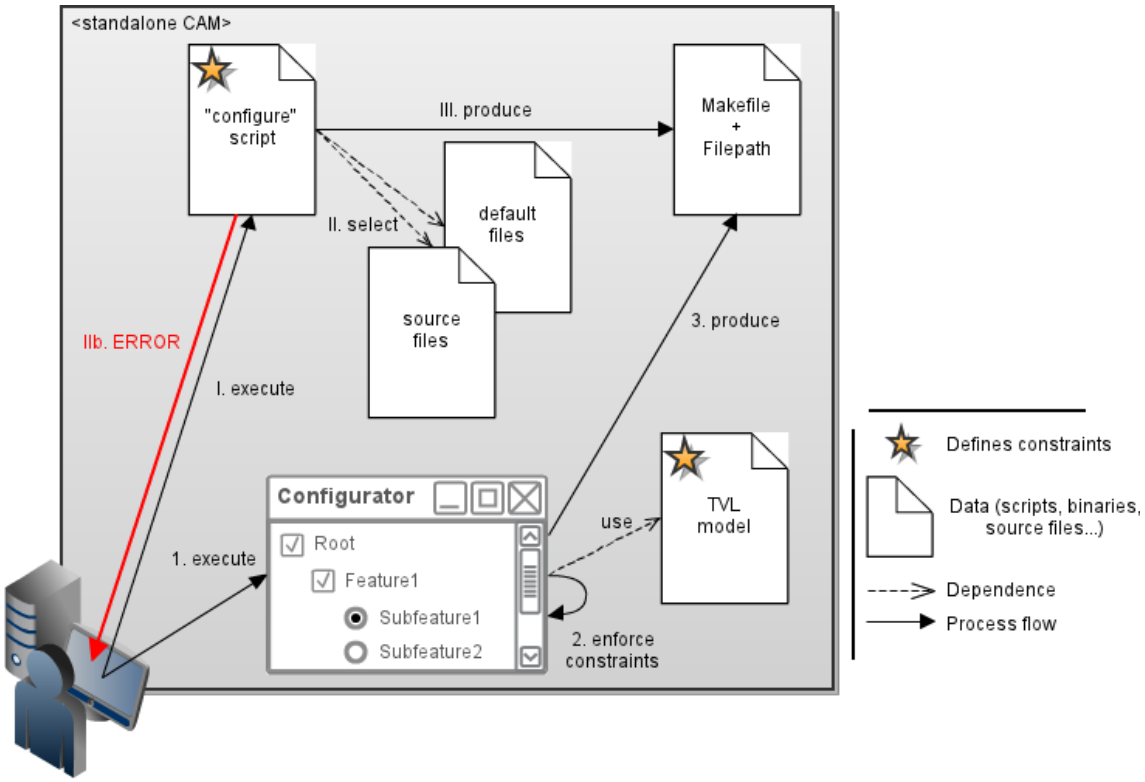


Figure 5.1.1: Descriptive diagram of CAM configuration.

[2]Makefile is used by **Make** utility to automatically build executable programs.
[3]Filepath contains the list of source directory paths required to compile with **Make**.

current configuration process using the configuration script is the following:

I **execute** The user executes the configuration script with the list of parameters which represents the configuration as input.

II **select** If the parameters meet all constraints, then the script selects all appropriate source directories, eventually according to a default configuration file.

IIb **ERROR** Otherwise, when a constraint is violated, an *error* message is raised and the process is interrupted. The user have to restart from the beginning.

III **produce** Finally, the script produces a `makefile` and a `filepath`. These files are used in the build phase.

Suggested configuration process explained below shows only the basic usage of the feature-based configurator (see Figure 5.1.1):

1. **execute** The user executes the configurator and the GUI is displayed which shows the feature model defined in the TVL model. Then he or she can interactively create a configuration.

2. **enforce constraints** Each time a choice is made by the user (i.e. selecting or de-selecting a feature), the constraints are checked by a solver and some options are auto-selected or disable, according to the constraints defined in the TVL model.

3. **produce** When there is no more "unknown" options, the configuration is valid and can be used to produce a `makefile` and a `filepath`. These files are used in the build phase.

5.2 Overview

The suggested configurator system, is composed of two main parts. The first part is a generic configurator using FMs. The primary goal of this generic configurator is to automatically enforce the constraints defined in a TVL model and easily produce a valid configuration. The second part mananges outputs. So the system can be customized to produce desired outputs according to users' configurations. For instance, in the case of CAM, the configurator would manage a large part of the current configuration script that means they have the same semantics about the constraints defined on features and outputs could be identical. The GUI would make the configuration process easier than the current configuration script. In essence, users would be able to see the interactions between features.

5.2.1 Functionalities

The system would be composed of two subsystems: a configurator and a generator. The configurator creates or edits a configuration and checks a configuration. The generator generates several outputs as: 1. translated FMs from TVL to the format used by SPLOT, called SXFM; 2. FDs corresponding to TVL FMs; and 3. outputs according a configuration to build a product, here a CAM standalone model.

The following paragraphs describe the functionalities suited to CAM usage, depicted in Figure 5.2.1 and basically available in CAMelot.

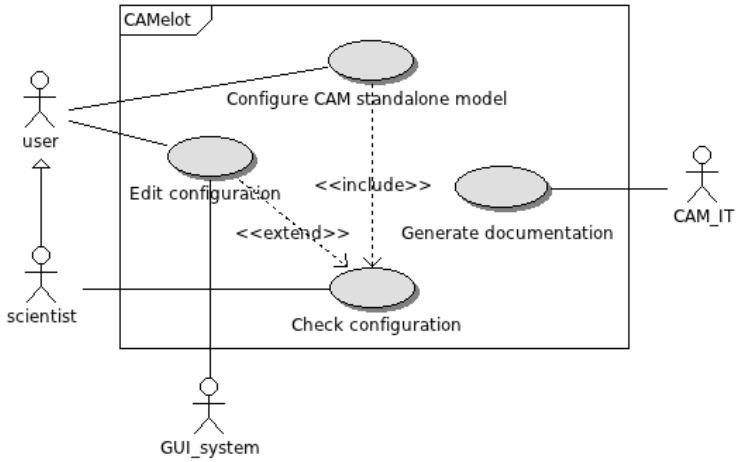


Figure 5.2.1: Use case diagram of the system.

Configure CAM standalone model Users can configure a CAM standalone model with CAMelot by providing a configuration and a TVL FM of CAM. CAMelot produces the required data to build the CAM model. Users could use the GUI_system to create or edit a `configuration` and provide it.

Edit configuration Users can edit or create from scratch a `configuration` with the CONFIGURATOR, according to a TVL FM. A GUI would be provided to make the process easier.

Check configuration Users, especially scientists, can submit a `configuration` to the CONFIGURATOR to check its validity according to a TVL FM.

Generate documentation Managers of CAM (CAM_IT) can automatically generate several documents based on a TVL FMs, like FDs, translation from TVL into SXFM or DIMACS format (with a mapping between IDs used in TVL and translated version).

5.2.2 Graphical user interface

This section describes a sketch of the GUI. To make it more explicit, it's based on the following hypothetical sample TVL model[4]. The root feature *MyRoot* is decomposed into two sub-features: the feature *MyFeature* and the optional feature *MyFeature2*. The feature *MyFeature* is composed of either the feature *Foo* or the feature *Bar*, and the feature *MyFeature2* is composed of the feature *Baz*. The feature *MyFeature* has also several attributes; those are an integer *myInt* which takes its value between 1 and 4, a real *myReal* which takes its value between 0.5 and 1, an enumeration *myEnum* with three choices (One, Two, Three), a Boolean *myBool*, a structure *myStruct* that is a couple of an integer *num* and a string *noun* where the value of *myStruct.num* is restricted to the value '253', and a string *myString* restricted to the value 'tadaam'. See Listing 5.1 p.48. This sample could be represented by an incomplete FD, as attributes aren't represented, made with FeatureIDE[38]. See Figure 5.2.2.

```
1 struct myStructure {int num; string noun;}
```

[4]Keep in mind that currently TVL doesn't support String attributes.

```
2
3  root MyRoot
4    group allof{
5      MyFeature
6        group oneof{
7          Foo,
8          Bar
9        },
10       opt MyFeature2
11         group allof{
12           Baz
13         }
14   }
15
16 MyFeature{
17   int myInt in [1..4];
18   real myReal in [0.5..1];
19   enum myEnum in {One, Two, Three};
20   bool myBool;
21   myStructure myStruct {num is 253;}
22   string myString is "tadaam";
23 }
```

Listing 5.1: myFeature TVL model

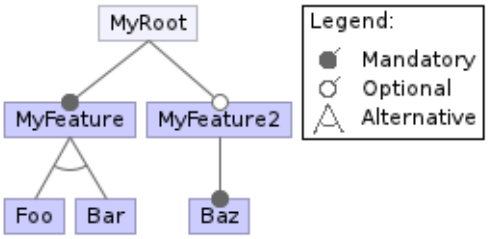


Figure 5.2.2: FD of “myFeature TVL model”

Figure 5.2.3 represents the primary window of the configurator, inspired by the LKC (Section 4.2.2) according to the above sample FM. This primary window is composed of the fundamental functionalities required to easily configure a product with a GUI. The left part of the primary window allows users to parametrize the hierarchical tree part of a FM. This part was actually created with the FeatureIDE configurator (Section 4.2.1). Each feature is represented by its name in a tree hierarchy as they are defined and is preceded by an icon that means the feature is selected (green cross), deselected (red minus) or not yet parametrized (empty square). On the top, users are informed of the validity of the current configuration and the number of remaining possibilities[5].

Furthermore, in TVL, each feature could have attributes that would be parametrized in the right part of the primary window. This part is divided into a notification and a parametrization areas. The notification area provides users with different kind of informative messages. The parametrization area allows users to see all attributes and edit the value of the enabled attributes. According to constraints in the TVL model, the value of some attributes can be restricted to an unique value, and so can not be changed.

Notice that menus are not represented and could host many functionalities like generate outputs according to the current configuration or open a configuration file.

[5]The number of remaining possibilities shown in Figure 5.2.3 doesn’t consider the attributes and their variability.

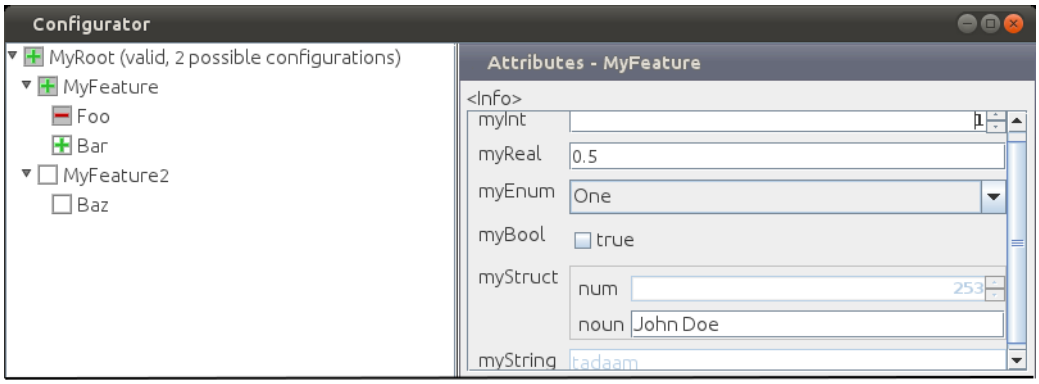


Figure 5.2.3: Primary window of the configurator.

5.3 Implementation

A prototype of the configurator system described in the previous section has been developed. This configurator, called CAMelot, is described in details in this section. CAMelot is more a tool to think about feasibility and interest to feature-based configurators with the application to a real case, and it's far to be a proof of excellence (CAMelot can not pretend to shelter the Holy Grail).

5.3.1 Working assumptions

CAMelot is a prototype, so some working assumptions limit its implementation. CAMelot has been developed to manage the configuration of CAM in the standalone mode. A TVL model was written according to the configuration script of CAM and the FD realized by R. Dunlap[6] (Appendix A).

Work on normalized TVL models. CAMelot uses the TVLparser to check the TVL model and get the model as an Abstract Syntax Tree (AST). Work on normalized TVL models, inside the configurator, simplify the management of cross-tree constraints as all feature IDs are global and local references are translated. However, we discourage local ID usage in the TVL model as we notice the management of long IDs is still work in progress in the TVLparser.

Limit to Boolean feature models. CAMelot only considers Boolean FMs that means only Boolean features and attributes are taken into account. Note that in TVL, enumeration attributes can be considered as Boolean attributes. Thanks to this limitation, we can easily manage validity of configurations with a SAT solver. We use the same solver used by the TVLparser (SAT4J 2.1). To support other types of attributes like integers, CSP solver are required. This kind of support raises some issues about the behaviour of the configurator when the domain of an integer attributes is not constrained, for example.

Use a directories map. One main purpose of the configuration script of CAM is the production of a `filepath` according to the features selected or deselected in the configuration. Each feature has basically a list of source directory paths that have to be used in the build phase.

[6] Available at `http://rockydunlap.files.wordpress.com/2010/11/cam_features.pdf`

As TVL currently doesn't support string attributes, and the implementation of data management by the TVLparser is work in progress, we have to use an external file to link the source directories to their associated features which is called *directories map*. This file is important to produce the `filepath`. As soon as either string attributes or data are fully supported the directories map could be integrated into the TVL model.

Generate partial outputs. The configuration script of CAM generates both a `makefile` and a `filepath` according a configuration. The `makefile` is used to configure **Make** according to the environment variables like the operating system. And the `filepath` lists the source directory paths that must be compiled with **Make** to build a CAM model according to the model variables. So the configuration is composed of two main types of variables: *environment* and *model* variables. We choose to focus our interest on the model variables and only take them into account, at first, in the CAM FM. Thus, CAMelot only manages model variables to produce the `filepath`.

Figure the configuration interface. Due to the limited time to develop the system, the configuration interface is not developed at first as it requires the knowledge of domain experts to suit appropriately their requirements. So the configuration edition activity is not yet supported by CAMelot. Instead, an online service named SPLOT[28] is currently used to figure the primary window of the configurator. SPLOT only manages basic Boolean FM that means Boolean attributes are not supported and have to be translated into features what is semantically equivalent. However, SPLOT doesn't support TVL language, thus the FM must be translated into the specific FM language used by SPLOT, called SXFM. That is the initial reason of the introduction of a translator component in our system.

5.3.2 Architecture

Concretely, a large part of the configurator system is generic, so it is implemented as a TVL tool, and CAMelot extends it to adapt the generation of outputs to build CAM standalone models. As previously said, the system is composed of two main components: a generic configurator to manage FMs and configurations, and a generator to manage outputs. Several additional components as a Translator, a Visitor and a TemplatesManager are used for specific managements. Each component is described in more details below and represented in Figure 5.3.1.

TVLparser is an external component. It parses the input TVL model and produces a normalized TVL model formatted into an AST.

GUI_system is an external component. It provides users with a GUI to interactively create a configuration based on an FM. Here, we currently use the configuration interface provided by SPLOT.

SATsolver is an external component. It resolves Boolean satisfiability problem and provides the set of solutions. Here, we use SAT4J that requires DIMACS CNF as input format to encode the problems (Section 4.1.2).

Configurator is composed by two sub-components: an Editor and a Checker; and uses a Translator. It manages configurations according to an FM.

Editor uses an Application Programming Interface (API) to communicate with a GUI_system which uses a SATsolver, and edit a configuration. This component is already present, even if currently the edition activity is not fully supported by CAMelot (Section 5.3.1) and requires the use of SPLOT.

Checker checks the validity of configurations that may be translated by the Translator and uses the SATsolver to support this checking according to an FM.

Generator uses the Visitor to walk through the AST produced by the TVLParser to produce translations of FMs. It also uses the Configurator to produce outputs according a configuration. All several outputs are described in the Section 5.3.5.

Visitor provides several visitors (see Section 5.3.4) to walk through the structure of a normalized TVL model produced by the TVLParser and uses a TemplatesManager to format it in another language syntax (i.e., SXFM).

Translator translates different configuration formats to a format that can be used by the Configurator. It also translates the FM from TVL into SXFM, DOT [16] or DIMACS format.

TemplatesManager manages several textual templates used by the Translator to translate FMs thanks to a template engine[7]. Here, we use StringTemplate (Section 5.3.5).

Utilz is an external component. It contains several utilities for basic management inside a system.

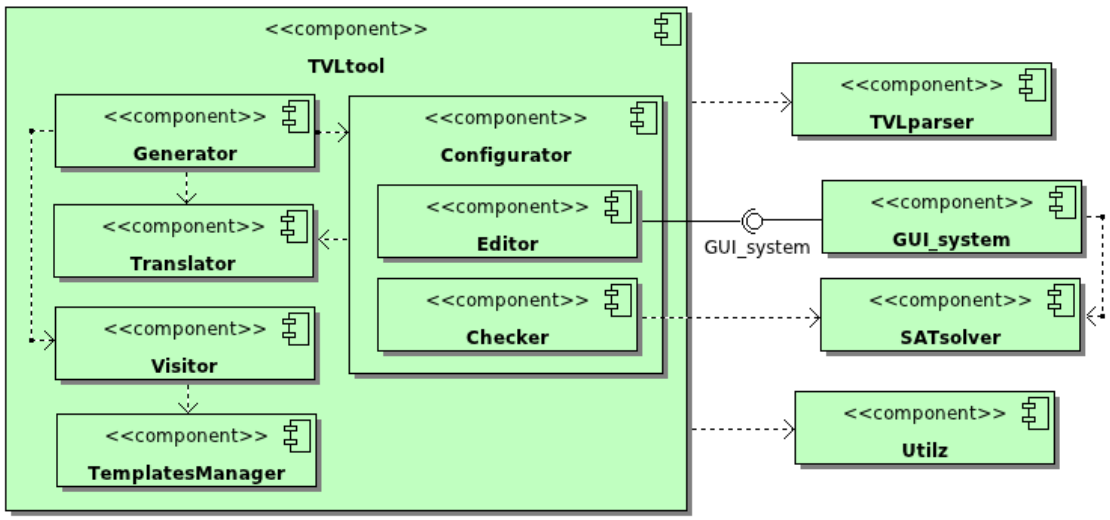


Figure 5.3.1: Component diagram

5.3.3 Configuration management

First of all, if we want to create a configuration according to an FM, this FM must be satisfiable. An FM is satisfiable if there is at least one combination of features that meets all the constraints. This condition can be easily checked for a Boolean FM with a SAT

[7]A template engine separates the process and its data structures from the textual output designs. It also provides the possibility of managing different designs without changing the process.

solver. One has just to translate the FM into the DIMACS format and passes it as input to the solver. This kind of translation and checking are supported by the TVLparser, and so CAMelot. Then when the FM is satisfiable, one can composed his or her own configuration according to a particular FM and checks its validity. CAMelot can check configurations composed as a list of TVL IDs of features that have to be selected or deselected[8], and configurations created with SPLOT saved into CVS or XML format.

To check a configuration, CAMelot initializes a SAT solver with the TVL model translated into DIMACS format and tests the configuration choices as additional constraints through assumptions of the solver. Similarly to the functionalities available with the configuration editor at SPLOT (Section 4.2.1), CAMelot can automatically complete a configuration by selecting/deselecting all remaining features.

5.3.4 AST browsing: Visitor pattern

The visitor pattern[18] is used in object-oriented programming and software engineering to separate an algorithm from an object structure on which it operates. In this way, new operations can be added to existing object structures without modifying those structures. However, the visitor pattern is more limited than conventional virtual functions. It is basically not possible to create visitors for objects without adding a small callback method inside each class, because of the double dispatching, as described below.

Details

> A user object receives a pointer to another object which implements an algorithm. The first is designated the "element class" and the latter the "visitor class." The idea is to use a structure of element classes, each of which has an *accept()* method taking a visitor object for an argument. *visitor* is a protocol (interface in Java) having a *visit()* method for each element class. The *accept()* method of an element class calls back the *visit()* method for its class. Separate concrete *visitor* classes can then be written to perform some particular operations, by implementing these operations in their respective *visit()* methods.(See Figure 5.3.2)
>
> [...] The visitor pattern also specifies how iteration occurs over the object structure. In the simplest version, where each algorithm needs to iterate in the same way, the *accept()* method of a container element, in addition to calling back the *visit()* method of the visitor, also passes the visitor object to the accept() method of all its constituent child elements. [18]

In Java, since Java 1.2, we can use the *reflection* mechanism [40, 9]. Thanks to this, visited Objects no longer have to be modified and ignore being visited, so the *accept()* method disappears. And any object becomes visitable, if a default method is added.

The visitor pattern with the reflection mechanism seems to be the best practice to manipulate an AST that corresponds to the model inside the generator, and produce the different output files from the generator according to the input TVL model.

5.3.5 Outputs generation

CAMelot produces several outputs through the generation activity. It can generate the translation of TVL models into SXFM or DIMACS format, these format are respectively

[8]The supports of attributes is not yet implemented.

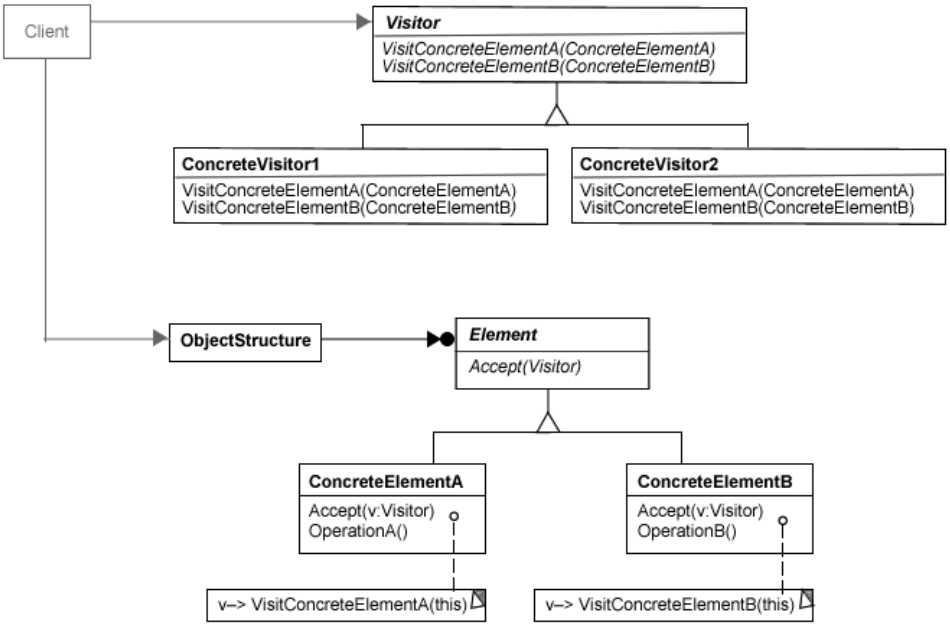


Figure 5.3.2: Visitor design pattern diagram.[18]

used by SPLOT and SAT solver. The translation of TVL models comes with a mapping between feature IDs used in both models. CAMelot can also create FDs corresponding to TVL models.

Model translation into SXFM format. TVL is a well-formed language with a formal semantics. So it is easy to defined syntactic equivalences with another FM format[9] as SXFM used by SPLOT. SXFM format supports only basic FM, so it's equivalent to a subset of TVL. The following example shows the main syntactic equivalences between TVL and SXFM languages. Listing 5.2 p.54 represents an FM sample where the root feature A is decomposed into two mandatory sub-features B and C, and an optional sub-feature D. The feature B is either I, J or K and has two attributes: the enumaration efg that takes its value in the set composed of e, f and g; and the Boolean b. The feature C is composed of a subset of the sub-features set of M, N and O. Four additional cross-tree constraints are also defined: 1. the selection of M implies the selection of J or I or both; 2. the selection of K implies the selection of O and vice versa; 3. the selection of J excludes the selection of N; and 4. the selection of I requires the selection of D.

```
1  root A{
2    group allof{
3      B group oneof{I,J,K},
4      C group someof{M,N,O},
5      opt D
6    }
7    M -> (J || I);
8    K <-> O;
9    J excludes N;
10   I requires D;
11 }
12
13 B{
```

[9]Section 3.3.2

```
14    enum efg in {e,f,g};
15    bool b;
16  }
```

Listing 5.2: Sample in TVL

The SXFM format is an XML-based language. It represents FMs as an indent tree with additional cross-tree constraints in CNF. Each feature is declared with the syntax `:<type>? <name> (<id>) [1,*]?` where `<type>` is either `r` (root), `m` (mandatory), `o` (optional), `g` (group) or *empty* (sub-feature of a group feature). The *group* type is used to represent *or*- and *xor*-decomposition, while the *and*-decomposition is defined as the default decomposition. So, the group type has an interval definition: `[1,1]` (*xor*-decomposition) or `[1,<nbSubFeature>]` (*or*-decomposition). `name` is the name of the feature, it could basically be anything, but it is encouraged to use meaningful names. `id` is the technical ID of the feature that is used in the additional cross-tree constraints. Constraints are declared with the syntax `<constraint_id> : ~?<feature_id> (or ~?<feature_id>)*`

```
1  <!-- This model was created to be used by
       SPLOT's Feature Model Editor (http://
       www.splot-research.org) -->
2  <feature_model name="Feature model of A">
3  <meta>
4  </meta>
5  <feature_tree>
6  :r A(feat_A)
7    :o D(feat_D)
8    :m B(feat_B)
9      :o B.b(bool_B_b)
10     :m B.efg(enum_B_efg)
11       :g (g_1) [1,1]
12         : B.efg_e(enum_B_efg_e)
13         : B.efg_f(enum_B_efg_f)
14         : B.efg_g(enum_B_efg_g)
15     :g (g_2) [1,1]
16       : I(feat_I)
17       : J(feat_J)
18       : K(feat_K)
19   :m C(feat_C)
20     :g (g_3) [1,3]
21       : M(feat_M)
22       : N(feat_N)
23       : O(feat_O)
24 </feature_tree>
25 <constraints>
26 constraint_1:~feat_M or feat_J or feat_I
27 constraint_2:~feat_K or feat_O
28 constraint_3:~feat_O or feat_K
29 constraint_4:~feat_J or ~feat_N
30 constraint_5:~feat_I or feat_D
31 </constraints>
32 </feature_model>
```

Listing 5.3: Sample in SXFM

Attributes are not supported in SXFM. However, thanks to the semantic equivalence, we can define a Boolean attribute as an optional feature and an enumeration attribute as a *xor*-decomposition. The following Table 5.1 highlights equivalent syntactical structures between the sample in TVL and SXFM (Listing 5.2 p.54 and Listing 5.3 p.55 respectively).

As SXFM is a XML format, it can be managed by a *template engine*. One example is StringTemplate [30] which is used by CAMelot. StringTemplate is a very powerful Java template engine. It allows the separation between the process and the output design. That means that we can define many different templates without changing anything in

Table 5.1: Syntactical equivalences between TVL and SXFM.

Structure	Sample in TVL (Listing 5.2 p.54)	Sample in SXFM (Listing 5.3 p.55)
Root feature	`1 root A{`	`6 :r A(feat_A)`
Mandatory feature	`2 group allof{`	`8 :m B(feat_B)`
Optional feature	`5 opt D`	`7 :o D(feat_D)`

And-decomposition	`2 group allof{`
	`3 B group oneof{I,J,K},`
	`4 C group someof{M,N,O}},`
	`5 opt D`
	`6 }`
		`7 :o D(feat_D)`
		`8 :m B(feat_B)`
		`9 :o B.b(bool_B_b)`
		`10 :m B.efg(enum_B_efg)`
		`11 :g (g_1) [1,1]`
		`12 : B.efg_e(enum_B_efg_e)`
		`13 : B.efg_f(enum_B_efg_f)`
		`14 : B.efg_g(enum_B_efg_g)`
		`15 :g (g_2) [1,1]`
		`16 : I(feat_I)`
		`17 : J(feat_J)`
		`18 : K(feat_K)`
		`19 :m C(feat_C)`

***Or*-decomposition**	`4 C group someof{M,N,O}},`	`20 :g (g_3) [1,3]`
		`21 : M(feat_M)`
		`22 : N(feat_N)`
		`23 : O(feat_O)`

***Xor*-decomposition**	`3 B group oneof{I,J,K},`	`15 :g (g_2) [1,1]`
		`16 : I(feat_I)`
		`17 : J(feat_J)`
		`18 : K(feat_K)`

Attributes	`14 enum efg in {e,f,g};`	`10 :m B.efg(enum_B_efg)`
		`11 :g (g_1) [1,1]`
		`12 : B.efg_e(enum_B_efg_e)`
		`13 : B.efg_f(enum_B_efg_f)`
		`14 : B.efg_g(enum_B_efg_g)`

	`1 bool b;V15`	`9 :o B.b(bool_B_b)`
Constraints	`7 M -> (J || I);`	`26 constraint_1:~feat_M or feat_J or feat_I`
	`8 K <-> O;`	`27 constraint_2:~feat_K or feat_O` `28 constraint_3:~feat_O or feat_K`
	`9 J excludes N;`	`29 constraint_4:~feat_J or ~feat_N`
	`10 I requires D;`	`30 constraint_5:~feat_I or feat_D`

the process. It can be used for any formatted text output and uses an uncomplicated language to write the templates. The template of SXFM format is described in more details in the Appendix B.2.

Model translation into DIMACS format. The translation into the DIMACS format is already supported by the TVLparser. It uses the Boolean form of the TVL model and produces the DIMACS file with the associated map between TVL and DIMACS IDs.

Feature diagram generation. CAMelot can generate FDs corresponding to any TVL models. We have extended the FODA notation (Section 3.2) to integrate attributes in the diagram. Attributes of a feature are inserted below the node of this feature with their type and ID. The cross-tree constraints are listed in the left-bottom corner, and so, don't really take part of the diagrams similarly to FDs displayed in FeatureIDE (Figure 4.2.2a). Figure 5.3.3 represents the FD of the sample in TVL (Listing 5.2 p.54).

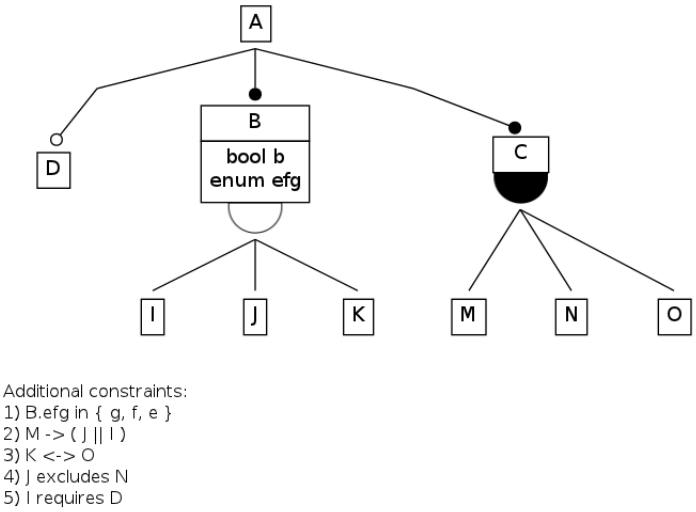


Figure 5.3.3: FD of the sample in TVL

CAMelot generates FDs thanks to DOT language [15] and so requires a program like *dot* to process the graph and get an image. As DOT is a text-based language, we use a template engine (StringTemplate) as for the model translation into SXFM format. The template of DOT format is described in more details in the Appendix B.3.

Data exchange Considering the main activity that is the configuration a CAM standalone model according an FM representation of CAM to produce the resulting `filepath`, let us make the data flow a little bit more explicit with the following description, and its representation in Figure 5.3.4. CAMelot receives an FM encoded in TVL, and the Generator produces a translated model in the format supported by the GUI_system (SXFM format) and a `feature map` that links IDs from TVL and translated models. These outputs are produced only if the model is satisfiable[10]. Then a GUI_system (SPLOT) using a SAT solver (SAT4J) shows and enforces constraints expressed in the FM to produce a configuration. The Configurator takes and checks the configuration to pass a valid and understandable configuration to the Generator. Note that a configuration created with

[10]A model is satisfiable if at least one configuration exists in which all constraints are met.

SPLOT is supposed to be already valid, and the checking is redundant. However, users could handwrite a configuration file or alter the configuration file from SPLOT to create a new configuration and pass it to CAMelot. In those cases, the configuration have to be checked. Finally, the Generator selects the list of source directory paths from the directories map[11], according to the configuration and produces a `filepath`.

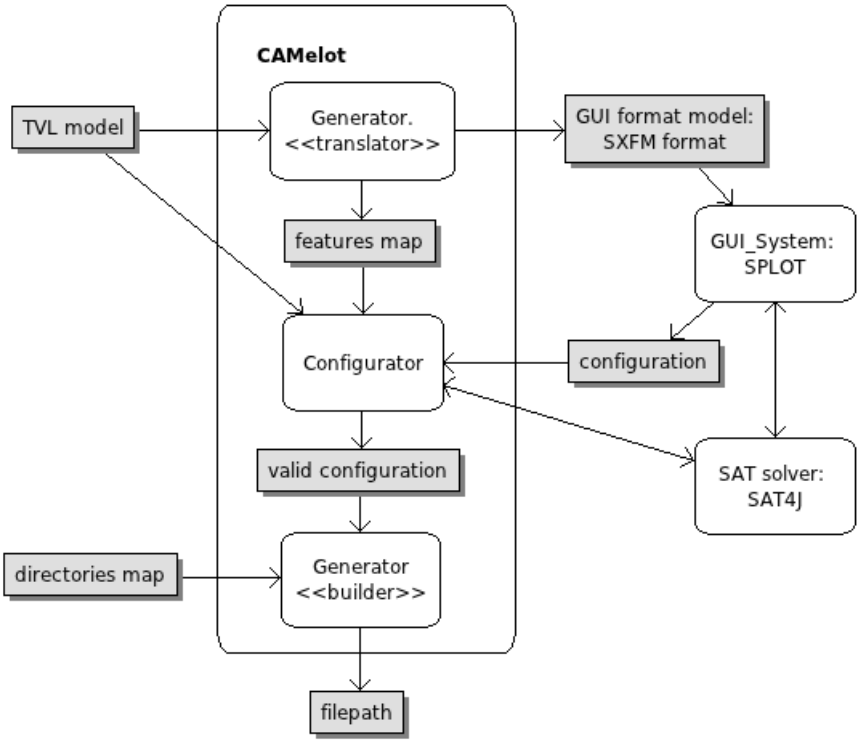


Figure 5.3.4: Data flow diagram of the system.

5.4 Evaluation

Our objective was to test the benefits of a feature-based configurator on a real case. CAM is a perfect study case as it contains an hundred of options both Boolean and numerical with some complex additional constraints. After the retro-engineering of the CAM configuration script, and the production of the corresponding CAM FM, we were able to determine if FMs can meet the challenges imposed by CAM (Section 2.6). Even if we have restrained our study to Boolean options, the potential of FMs can not be under-estimated. However, integration of numerical options may affect the performance. This kind of drawback is not discussed in this thesis, even if we are well aware about its importance in practice. We have preferred focus the study on a more abstract level of thought.

5.4.1 Meeting the challenges

- **Improving maintainability of CAM system.** (*Challenge C1*)
  Our work shows that FMs can improve the maintainability of CAM. Especially using a text-based feature modelling language like TVL. This language is concise and

[11] Until TVL fully supports string attributes or data, this map must be handwritten.

scalable thanks to the modularisation mechanisms. The definition of the interactions between CAM options is really easier to understand thanks to the readability of the TVL model. Indeed, constraints between options are defined declaratively or translated into sibling relationship if they are related. Moreover, this definition is independent on their definition order, unlike inside the configuration script. Plus, the consistency of the CAM definition is easily checkable with a dedicated tool like a SAT solver in our case.

- **Setting up a separation of concerns.** (*Challenge C2*)
  According to A. Hubaux [19, Chapter 4], FMs can set up a separation of concerns. The separation between definition and checking of constraints is well support as we have on one hand an FM that defines the constraints among options, and on the other hand a tool to perform various checks like consistency, and build a configuration. Thus programmers can easily maintain the system and work according to the concern of a dedicated task. On the point of view of users, they can easily configure a CAM model without taking care of what programmers do, they can focus themselves on their area of knowledge which is typically climate interactions modelling in this case. In our study, only a subset of options managed by the configuration script are considered, that is the climate model options. Thus, our CAM FM represents the variability inside a climate model. The other options for machine definition, and archiving can be considered in other FMs. Then these three FMs could be unified inside an unique FM thanks to modularity mechanisms like **include** in TVL. This is basically transparent for users but can be really useful for programmers as they can distribute the modelling task to domain expert without need of a configuration script expert to keep it running.

- **Highlighting options interactions.** (*Challenge C3*)
  R. Dunlap has figured out a lack of documentation about interactions between options. FDs are a good support to document this. Its visual notation lets users embrace immediately the interactions of related options, and modelling errors are also easier to detect as a graphical notation is generally more concise than a semantically equivalent textual notation. However, FDs are less scalable for expressing large FMs. So the choice between a graphical and a textual notation depends on the scale of the model, and its usage. Anyway, we integrate a functionality to generate FDs corresponding to TVL FMs in our prototype. Moreover, a configurator interactively highlights the influence of user's choices. Thus, the trial-and-error approach to configure a model would disappear, especially the error phase.

- **Increasing accessibility.** (*Challenge C5*)
  Increase the accessibility of CAM is not really require by its community, as they have learned to deal with its complexity. However, we think that it could have its importance to let the community focus on its initial purpose without taking care of technical matters. As the separation of concerns (*Challenge C2*) and the highlighting of interactions (*Challenge C3*) are already met, we could affirm that users can easily configure a CAM model with only climate knowledge. The configurator has to enforce the constraints defined inside the FM, and this model represents the current domain knowledge. According to these assumptions, if a model configured with the configurator is invalidated by climate researchers, they could easily assume that the FM has to evolve. As soon the modification is integrated, they could be sure that they will not be able to recreate an equivalent CAM model.

- **Supporting domain and model evolution.** (*Challenge C6*)
  Support domain and so model evolution, as the model represents the current climate domain knowledge, requires a lot of attention and effort. However, thanks to the readability of TVL FMs, it would be easier and straightforward than it is currently with the configuration script. Moreover, as TVL models are text-based, version management could be apply, and they still easily shareable.

- **Supporting default options.** (*Challenge C4*)
  This work was presented to the managers of CAM. They were enthusiastic about the potential of feature modelling. They also mentioned their need of default values to simplify the configuration by users. The support of default values is currently not part of FM. Thus, at a first stage our work omitted the default value support. As this requirement was already present in the feedback received during TVL study cases [20]. We'll suggest ideas to integrate default values in FMs and an TVL extension in the next chapter.

5.4.2 Disadvantages

Our work shows that FMs could meet most of the challenges offered by CAM, what is a great deal. However, even if a feature-based configurator has benefits for CAM, it introduces the disadvantages of FMs and solvers used to automatically enforce constraints.

Remind the reader of main disadvantages previously mentioned below:

- The scope of TVL features might rapidly become hard to identify. So know whether the referenced feature in constraints is unique or if a relative path has to be given is tricky.

- Missing constructs like defaults values in FMs could limit the representation of the domain knowledge.

- Solvers are generally resource-intensive. For example, SAT solvers are time-consuming and BDD solvers are space-consuming. So their use must be carefully scheduled and optimized.

In our study, we avoid these drawbacks thanks to lucky opportunities. We have used global IDs inside the TVL model. As the options are related but really different they can be easily identified with an unique ID. We had to deal with the missing constructs, so we introduced some working assumptions like only Boolean FM support, and the default values was not consider at first. Extend the FM to support numerical attributes, for instance, implies a wide modification in the "configurator kernel" that means a CSP solver would have to be integrated to continue to automatically enforce the constraints. Finally, the drawback of solvers do not really occur during our study as the CAM FM still pretty small and we benefited from the SPLOT expertise to manage their resource consuming.

Chapter 6

Default values

Currently, customers can customize most of products and services before purchasing them. Computer systems, automobiles, insurance policies and computer software, for examples, include many possible options that the costumer can select to create its own configuration. Configurators are typically provided to customize and configure this solution. Some of these configurators are constraint-based. Constraints are enforced between optional choices, allowing users to select the choices they want, while validating that the resulting set of user choices is valid [5].

Configure a system with number of options may easily become a burden for users. They may not know what decision to make, or care about the selection of every option required to complete the configuration. Make the process easier and faster for users is required by all business. One way is the usage of "default values", also called default decisions.

> A *default decision* may be a simple choice (i.e., a value that is assigned to some variable) or it may be a more complex constraint involving one or more variables as operands. [5]

According to C.M. Bagley et al., in the known constraint-based configurators, the default values are processed as decision after all user choices have been made. However, users are not able to see the default values or their consequences while making their own decisions. Plus, if users finally want to change the default values, the entire configuration has to be done again from scratch [5].

This kind of default structure is currently not part of FMs as it does not directly influence variability modelling in its fundamental definition. But integrate them earlier in the configuration process, at the modelling stage, would inform users about influence of default values on user choices. They may stand as primary configurations (even partial) that suggest values to users or complete the configuration after all user choices have been made. That is the problem to know when introduce these values in the configuration. Moreover, an interesting behaviour for default values could be that they take their values according to some user choices. This conditional behaviour is more complex, but really powerful to represent some real business needs. For instance, a customer wants to purchase a car, the colour of the car can depend on the brand like if it's a *Ferrari* the default color is *red* and if it's a *Lamborghini* the default color is *yellow*.

In every cases, default values are equivalent to decisions but considered after user's choices. That means user's decisions always override default values. Concretely, if no user's choice is made for a variable then a default value is considered, but if no default

value is defined then the user have to make a decision to complete the configuration. In this chapter, we'll propose three views[1] of default values inside a feature-based configurator. This integration rises up an important question about the consistency of FMs supporting default values. Finally, we'll suggest a theoretical TVL extension for default values and apply it to the CAM case.

6.1 Integration of default values

This section aims to present some ways to integrate default values into a feature-based configuration process, according to the meaning of these values, their influence on a configuration and their technical support. Three main categories appeared during our study: 1. initial configuration; 2. fill-in configuration; and 3. conditional default values. Each category will be defined, described through its influence on a basic configuration[2] and evaluated. Default values could be stored whether directly inside the FM or in a dedicated component like a default configuration file, both approaches are discussed in the sub-sections below.

6.1.1 Initial configuration

Default values considered as atomic pre-defined decisions is the simplest way to understand them. That means that before users make any decisions, a particular value is already chosen for some variables and can be assimilated to a suggestion to the user to help him or her to configure a product. The set of default values is called an *initial configuration* in this case. If default values are equivalent to suggestions, users must be able to change the value assigned to a particular variable. For example, if a feature is selected by default, users can chose to deselect it. Obviously, each decision must continue to meet the constraints defined in the FM, decisions propagation must be applied on default decisions as well on user's choices. Otherwise the configuration would become invalid. The default configuration could be whether partial if some variables have no default value or complete if all variables have an assigned default value. A complete initial configuration is equivalent to a valid configuration previously made that users eventually want to edit.

The most important part of the implementation is the integration of default values during the configuration process. Initial configuration should be instantiated at the beginning of the process, *before* all user's decisions. Default decisions should be considered as *assumptions* like user's decisions and marked to be able to reset them if the user change his or her mind after a customization. Thus they can be displayed as user's choices with the possibility to differentiate them.

Storage of default values

Several solution exist to store initial configurations, both partial and complete. Depending on the number of initial configurations provided to users, they could be directly encoded inside FMs or stored in a dedicated component like a default configuration file. When the domain offers only one initial default configuration, the best storage should be to encode it *directly into the FM*. To do so, informally, we could use the keyword **default** followed by the default value assigned to a particular FM structure. A TVL example of a possible syntax is shown below with a short description for each main group of FM structures.

[1] These views are influenced by the work of C.M. Bagley et al. [5] and the Linux Kernel Configurator [35]
[2] A basic configuration is created without default values.

Optional features can be *in* or *out* the configuration by default. In TVL, it could be encoded as `A{group allOf{B, opt D} default {D;}}` that means the optional feature D is selected by default.

Cardinality-based decompositions that also consider *or*- and *xor*-decompositions can take a set of feature as default value. In TVL, that could be `A{group someOf{B,C,D} default {B&&C;}}` which means the feature A is composed by default of the sub-set {B,C} of the set of feature {B,C,D}. Another example, `V{group oneOf{X,Y,Z} default {Y;}}` means that the feature V is composed by default of the sub-set of an unique feature {Y} from the set {X,Y,Z}.

Attributes can be assigned to a particular value. In TVL already exists the **is** keyword that restricts the value of an attribute to a particular value. Assign a default value follows the same idea without the strict restriction, the default value is a decision that can be overridden unlike constraints. For instance, `Real pi default {3.14;}` means the real attribute *pi* takes the value 3.14 by default, but the user can change it to 3.1415 if the user want a more precise value of pi.

Define defaults value with this kind of storage is straightforward. However, it does not provide a global view of the default configuration, and conflicts between default value can easily appear.

When many different default configurations can be provided, a better storage is to encode them into *default configuration files*. As their name suggests it, a default configuration file contains a list of pre-defined decisions to create a configuration. In fact, it is a configuration file with the list of selected and deselected features that an user can edit. The concrete syntax of configuration files depend on the configurator. This kind of storage is more generic than the previous one and does not modify the FM. However it is generally less readable, and conflicts between default values are also hard to detect inside configuration files.

Benefits & Disadvantages

Thanks to suggestion aspect and pre-defined decisions of initial configurations, they could decrease significantly the time required to configure a product with a lot of options. Further, several initial configurations can be defined for a particular model. A complete initial configuration provides a valid configuration (assuming there are no conflicts between the values) and indicates there is at least one valid configuration.

However, complete initial configurations could lead to a lack of interest from the user to the configuration, and make the system useless. Partial initial configurations could have only few variables without default values, in this case users could have some difficulties to know them, and provide a mechanisms to highlight the decision points should be a great idea. Plus, users always have to take part of the configuration process based on a partial default configuration.

6.1.2 Fill-in configuration

Fill-in configuration represents a set of default values used when the user don't make any decisions about some variables. It's really similar to initial configuration, unless default value are considered *after* all user's choices have been made. So compared to initial configuration, default decisions are no longer like suggestions, but could be used as an *auto-completion* for the current user's configuration. This auto-completion should be

marked to let the user know the modification, and let him or her modify any attributed default values that do not meet his or her desire. This is the mechanism used by most of known constraint-based configurators according to C.M. Bagley et al. [5].

The storage techniques of default values are the same as those used for the initial configuration described above. Even if in this case provide several different default configuration is less relevant. Thus the storage inside the FM should be preferred.

Benefits & Disadvantages

In this case, users are not influenced by the default values and can freely make their configuration. So users have a real role during the configuration process.

However, users can not trust that default values will configure all variables they don't parametrized. Especially, if the fill-in configuration is partial that means some variables have no default value. Another case is a conflict between an user's choice and a default value that should be apply, in this case the default value would be ignored, the variable is not configured, and the user have to make a decision to complete the configuration.

6.1.3 Conditional default values

This sub-section presents a more powerful type of default values, called *conditional default values*. It can be used as initial or fill-in configurations that can be considered as particular cases of conditional default values. But its real power is the possibility to define default values *according to* user's choices. Remind the previous example of a customer that wants to purchase a car, the colour of the car can depend on the brand like if it's a *Ferrari* the default color is *red* and if it's a *Lamborghini* the default color is *yellow*. An initial default value could be provided as well to let the possibility to the user to make no choice. Of course, some variables could have no default values at all or no values that meet the user's decisions. In that case, the user have to make a choice to complete the configuration.

Default decisions should be considered as *assumptions* like user's decisions and checked after every user's decision to fit their conditional aspect. However, it rises up issues about which assignment should be apply on a variable already configured by the user when the condition of a conditional default value is met. Keeping the previous example, the user configures the colour *black* for his or her car and then sets up the brand to *Ferrari*; the issue is 'which colour is the car?', the answer *black* or *red* would depend on the implementation of the configurator, but we suggest to put the priority on the user's choices, so the colour should stay *black* in our example. As user's and default decisions could appeared at the same step through decision propagation, a specific display notation should be provided by the configurator to differentiate them.

Storage of default values

Conditional default values should ideally be stored inside FMs as it is really similar to constraint definition without the restriction aspect. To do so, informally, we could use the keyword **default** followed by a block of the conditional assignment of default value to particular variables according the value of some others. Thus the definition of default values can become as complex as constraint definition. A TVL example of a possible syntax is shown below with a short description for each main group of FM structures.

Optional features can be *in* or *out* the configuration by default. In TVL, it could be encoded as `C{allOf{E, opt D} default {D; A||B->!D;}}` that means the optional

feature D is selected by default, but it is deselected by default if the user chose at least A or B.

Cardinality-based decompositions that also consider *or*- and *xor*-decompositions can take a set of feature as default value. In TVL, that could be `A{group someOf{B,C,D} default {U->B&&C;}}` which means the feature A is composed by default of the sub-set {B,C} if the user selected the feature U. Note that D stays undefined. Another example, `V{group oneOf{X,Y,Z} default {Y; U&&T->X}}` means that the feature V is composed by default of the sub-set of an unique feature {Y} from the set {X,Y,Z}. But if the user selected the features U and T, V is composed by default of the sub-set of an unique feature {X}.

Attributes can be assigned to a particular value. For instance, `Real pi default { 3.14159265; !Scientist->3.14;` means the real attribute *pi* takes the value 3.14159265 by default, but if the user deselected the feature *Scientist* then *pi* is set to 3.14 by default.

Keep in mind that all default values can be overridden by a user's decision. They are not equivalent to constraints.

Benefits & Disadvantages

Conditional default values are way more powerful than other definitions of default values. It can be used to define complex default definitions that is closer to real cases.

However, it increases the complexity of the model, especially its management by a configurator. It becomes also impossible to detect conflicts between default values in a large scale model.

6.2 Extended model and configuration consistency

Consistency is a critical point in FMs, if a model is not satisfiable that means whether the domain is not well understand or there are some mistakes in the model encoding. Hopefully, integrate default values into FMs does not influence the model consistency. They only acts on configurations. So configuration consistency must be ensure regardless of the type of default values that is used. The following paragraphs describe a way to ensure the consistency of configuration along the process. Some of these ways may be resource-intensive and should require an optimization to be implemented.

Initial configuration

In initial configurations, default values are instantiated inside the current configuration before any user's decisions. During the configuration, at each user's decision, the set of default value has to be checked. Indeed, a default value may be overridden that means the user made a decision for the structure related to the default value, or some defaults values could enter in conflict with the user's choice. To perform the checking, all default values are activated. Then in the cases described above, the defaults value are removed from the current configuration that means the default values are deactivated. All activated default values are finally displayed to help the user to complete the configuration. Thus the configuration is still consistent, but in the extreme case default values are all removed and so become useless. The consistence of the set of default values could be checked on the

FM without performing the configuration process, as this default set could be used as a (partial) configuration. At any time, user's and default decisions are managed separately and easily identified.

Fill-in configuration

Fill-in configurations acts similarly to initial configuration. The difference is the number of default value checks. At the beginning and during the configuration, the user has no idea of default values. Default values are inserted into the configuration when the user wants to auto-complete his or her configuration. The process is the same as for initial configuration. All default values which are activated, then the consistency of default values is tested based on user's decisions (translated into an assumption) and the model. If a default value rises a contradiction or is overridden by a user's choice, this value is removed from the current configuration. If the configuration is complete the process is over. Otherwise, default decisions are assimilated to user decisions, and default values are not checked until the user assesses again that he or she has finished his or her configuration. This type of default values is less ressource-intensive, and the configuration stays consistent, but users do not really know about default values. The consistence of the set of default values could also be checked on the FM without performing the configuration process. But it is less useful than in the previous case, as the chance to use this default set as a (partial) configuration is very low.

Conditional default values

Conditional default values are instantiated inside the current configuration before any user's decisions as in the case of initial configuration. During the configuration, at each user's decision, the set of default value has to be checked. To do so, all default values which the condition is satisfied are activated. Then the consistency of default values is tested based on user's decisions (translated into an assumption) and the model. If a default value rises a contradiction or is overridden by a user's choice, this value is removed from the current configuration. All activated default values are finally displayed to help the user to complete the configuration. Thus the configuration is still consistent, but in the extreme case default values are all removed and so become useless. At any time, user's and default decisions are management separately and easily identified. Consistency of conditional default values can not be checked without performing the configuration process as it all depend on the user's decisions.

6.3 TVL extension

In this section, the specification of a TVL extension to support conditional default values inside a FM is described. We provide the grammar extension and its semantic, and we finish by an application of this language extension on a real case: CAM.

6.3.1 Syntax

To define a grammar extension to support default values into TVL, we'll follow the same principles defined in the TVL specification [12, Section 3], where the reader can find the current grammar specification. Remind the principles:

> The grammar is a conflict-free LALR grammar and is given in extended Backus-Naur form (EBNF): terminals are encoded in double quotes, parentheses are used for grouping, (S)? means S is optional, (S)+ means that S repeats one or more times and (S)* is a shortcut for ((S)+)?. To make the rules more readable, non-terminals are written in uppercase. [12, Section 3]

Extend feature declarations

The feature body already consists of several items which can be data blocks, constraints, attributes or the group block declaring the child features. We extend this definition and add a default block that defines the conditional default values.

```
FEATURE_BODY_ITEM = [...]
                  | DEFAULTS ;
```

Extend attributes declarations

The attribute body allows to restrict the domain of an attribute, or to give it a value as part of the attribute declaration (instead of doing it in the constraints). The keywords **in** and **is** are used respectively on these purposes. Following the same idea, we extend this definition with a default block to suggest conditional default values as part of the attribute declaration (instead of doing it in the defaults blocks of features).

```
ATTRIBUTE_BODY = [...]
  | ("in" SET_EXPRESSION)? "default" "{" DEFAULTS_ATTR_LIST "}"
                           ("," ATTRIBUTE_CONDITIONAL )? ;
```

Add defaults declarations

A default block starts appropriately with the **default** terminal, followed by a list of default expressions. There can be several default block in each feature, all being merged when the model is parsed. Even if declarations of default values are written like constraints, default values do not constrain the model, they just suggest values to complete a configuration. Default values do not have any meaning in the normal FD semantics.

```
DEFAULTS = "default" "{" DEFAULTS_LIST "}" ;
```

An default list is just a list of default expressions separated by semi-colons. It is used to defined the body of default block in feature.

```
DEFAULTS_LIST = DEFAULTS_EXPRESSION ";" (DEFAULTS_LIST)*
              | EXPRESSION "->" DEFAULTS_EXPRESSION ";" (DEFAULTS_LIST)* ;
```

A default expression is a restricted expression that can only be composed of a long ID, a literal negation, a conjunction of default expressions or a value assignment.

```
DEFAULTS_EXPRESSION = LONG_ID
              | "!" LONG_ID
              | "(" DEFAULTS_EXPRESSION ")"
              | DEFAULTS_EXPRESSION "&&" DEFAULTS_EXPRESSION
              | LONG_ID "=" EXPRESSION;
```

A default attribute list is just a list of default declarations separated by semi-colons. It is used to defined the body of default block in attribute declarations. The left side of the implication is the condition to apply the default value assignment. The right side of the implication is the default value assignment to be applied on the attribute to which the default list is attached.

```
DEFAULTS_ATTR_LIST = EXPRESSION ";" (DEFAULTS_ATTR_LIST)*
              | EXPRESSION "->" EXPRESSION ";" (DEFAULTS_ATTR_LIST)* ;
```

6.3.2 Semantics

This sub-section expressed the semantics attached to the syntax introduced above to support default values into TVL. Declarations of default values have the following syntax:

$$Condition \rightarrow Default;$$
$$\text{or}$$
$$Default; \text{ (equivalent to } true \rightarrow Default;)$$

where

Condition is the condition to which the default value is applied, and it is expressed by an `Expression`[3]; and

$Default$ is the default value assignment expressed by an `Default expression` in feature blocks or a `Expression` in attribute declarations.

A `Default expression` D is formed according to the following grammar:

$$D ::= I \mid !I \mid D \;\&\&\; D \mid a = e$$
$$I ::= n \mid a$$

where

$n \in$ N (the non empty set of features) is a feature;

$a \in$ A (the set of attributes) is an attribute; and

e is an `Expression`.

Operator precedence, associativity and parentheses for default expressions have to be defined like for other expressions in TVL and TVL$_{NF}$. Operator precedence is defined to be the same as in TVL and TVL$_{NF}$.

Definition 1. *(Operator precedence in TVL and TVL$_{NF}$). Table 6.1 lists all operators in decreasing order of precedence. The associativity of each operator is given in left column. Parentheses can be used to group default expressions and clarify an evaluation order.*

Table 6.1: Operator precedence in TVL and TVL$_{NF}$

Associativity	**Operators**
right	!
right	=
left	&&
left	->

Definition 3 from the TVL specification [12, Section 5] exposes steps order to obtain a model in TVL$_{NF}$ from a model in TVL. Two additional steps have to be added to manage default values declarations. The first step, called **Attribute default value specifications**, should come right after the **Attribute domain and value specifications**. In this

[3] `Expression` is a TVL expression well-defined in the TVL specification [12]

step, the construct `t a default {v;};` allows to specify the default value of an attribute `a`. The `default` construct is removed from the attribute declaration, and a default value declaration of the form `this.a = v` is added. Similarly, the construct `t a default {c -> v;};` allows to specify the conditional default value of an attribute `a`. The `default` construct is removed from the attribute declaration, and a default value declaration of the form `c -> this.a = v` is added. And the second step, called **Default values**, should be added between steps **Constraints** and **Decomposition operators**, in order to obtain in TVL a single set of default values by moving all default values declarations to the root feature.

As noted before, default values does not concern explicitly FMs but more especially their instantiation through a configuration. Default values are equivalent to decisions that can be overridden by user's decision. They can be managed by a reasoner as constraints through assumptions. That means, according to Definition 4 from the TVL specification [12, Section 5], each product p is a couple $p = (c,v)$, where c is a set of features and v is a valuation of the attributes:

$$\forall \phi \in \Phi \bullet (Cond \rightarrow \phi \ \wedge \ [\![Cond]\!] \models true) \Rightarrow [\![\phi]\!](c,v) \not\models false$$

iff no user's decisions override ϕ.

A default value is overridden by user's decisions if

$$\forall \phi \in \Phi, (Cond \rightarrow \phi \ \wedge \ [\![Cond]\!] \models true) \bullet [\![\phi]\!](c,v) \models false$$

The semantics of a default expression, $[\![\phi]\!](c,v)$, is given in Table 6.2

Table 6.2: Default expression semantics in TVL_{NF}, that is, the value of $[\![\phi]\!](c,v)$

$[\![n]\!]$	=	*true* iff n ∈ c
$[\![a]\!]$	=	v(a)
$[\![!I]\!]$	=	*true* iff $[\![I]\!]$ equals *false*
$[\![D_1 \&\& D_2]\!]$	=	*true* iff $[\![D_1]\!] \wedge [\![D_2]\!]$
$[\![a = E]\!]$	=	*true* iff $[\![a]\!]$ equals $[\![E]\!]$

6.3.3 Example on a real case

As a concrete example, we apply the TVL extension to CAM. So we have extracted the definition of default values from the configuration script and written the following file (Listing 6.1 p.69) that encodes the default values in the syntax described above in Section 6.3.1. It was this case that brought to our attention the importance of conditional default values.

```
1  //# Default values definitions #
2
3  //**Physics package
4  Phys_pkg{ default {Cam5;}}
5  //**Chemistry package
6  Chem_pkg{
7    default {
8      Trop_mam3;
9      !Phys_pkg.Cam5 -> !Chem_pkg;
10     Chem_pkg.Waccm_ghg || Chem_pkg.Waccm_mozart
```

```
11        -> Dynamics.Fv && Phys_pkg.Cam4 && Hgrid.FvGrid.F1_9x2_5;
12      !(Chem_pkg.Waccm_ghg || Chem_pkg.Waccm_mozart) && Dynamics.Eul
13        -> Hgrid.EulGrid.E64x128;
14      !(Chem_pkg.Waccm_ghg || Chem_pkg.Waccm_mozart) && Dynamics.Sld
15        -> Hgrid.SldGrid.S64x128;
16      !(Chem_pkg.Waccm_ghg || Chem_pkg.Waccm_mozart) && Dynamics.Homme
17        -> Hgrid.HommeGrid.Ne16np4;
18      !(Chem_pkg.Waccm_ghg || Chem_pkg.Waccm_mozart) && Dynamics.FV
19        -> Hgrid.FvGrid.F1_9x2_5;
20    }
21  }
22  //**Interface
23  Interface{ default {Mct;}}
24  //**Microphysics
25  Microphysics{
26    default {
27      Phys_pkg.Cam5 -> Microphysics.Mg;
28      !Phys_pkg.Cam5 -> Microphysics.Rk;
29    }
30  }
31  //**PBL
32  Pbl{
33    default {
34      Phys_pkg.Cam5 -> Pbl.Uw;
35      !Phys_pkg.Cam5 -> Pbl.Hb;
36    }
37  }
38  //**Radiation package
39  Radiation{
40    default {
41      Phys_pkg.Cam5 -> Radiation.Rrtmg;
42      !Phys_pkg.Cam5 && (Chem_pkg.Waccm_ghg || Chem_pkg.Waccm_mozart)
43        -> Radiation.Camrt;
44    }
45  }
46  //**Ocean package
47  Ocean{
48    default {
49      Docn;
50      Phys_pkg.Ideal || Phys_pkg.Adiabatic -> Ocean.Socn;
51    }
52  }
53  //**Land package
54  Land{
55    default {
56      Clm;
57      Phys_pkg.Ideal || Phys_pkg.Adiabatic || Ocean.aquaplanet
58        -> Land.Slnd;
59    }
60  }
61  //**Sea ice package
62  SeaIce{
63    default {
64      Cice;
65      Phys_pkg.Ideal || Phys_pkg.Adiabatic || Ocean.aquaplanet
66        -> SeaIce.Sice;
67      Phys.pkg.Cam3 -> SeaIce.Csim4
68    }
69  }
```

Listing 6.1: Default values definition of CAM FM.

This file could be included into the TVL model if our suggested extension of language is integrated into TVL. Support default values also requires to implement their management into the configurator.

6.4 Summary

Default values are a requirement of the industry to help customer to easier configure products that can have a high number of parametrizable options. This variability can be modelled with FMs. But these models don't currently support default values.

Integrate default values in FMs rises up an important issue about their consistency. We have seen that default values don't actually acts on FMs but on their instantiation, the configurations. And the consistency of those configurations can be ensure by some mechanisms inside the configurator. Default values can be declaratively defined like constraints inside FMs, but configurators have to consider them like user's choices with a lower level of priority that means user's choices must always override default values when they are applied on the same variable.

Finally, this section has introduced a language extension to support conditional default values into TVL. However, because of a lack of time, this suggestion remains theoretical as no concrete integration of the language extension has been implemented yet. Only the syntax and its semantics are defined and presented in this thesis.

Chapter 7

Conclusion

In the first part of this thesis, the problem statement was introduced with its challenges in Chapter 2. The high level of variability in the configuration of climate models makes its management difficult. Moreover, its constant evolution does not help. Nonetheless, a solution was already pointed out by R.Dunlap, as he affirms the similarity between CAM and an SPL. According to this assumption, he produced a first FM based on the CAM configuration script.

Effectively, FMs are generally used to represent the key principle of SPLs: the variability. SPL paradigm was discussed in Chapter 3. Plus, concepts of variability modelling were described and their benefits and disadvantages were also exposed in this chapter. Two types of notation for FMs were presented: 1. graphical notation; and 2. text-based notation. Each notation has its benefits and disadvantages. However, a text-based notation with a formal semantics like TVL is more appropriate to perform automations on FMs. Another important concept inherent to FMs, their instantiation through configurations, was also discussed later in Chapter 4. In this chapter some examples of feature-based configurators like LKC were introduced and influenced our work.

In the second part, the prototype of a feature-based configurator based on TVL was described (Chapter 5). This prototypes was implemented during our internship to help our teamwork to figure out how feature modelling and related tools could be helpful for CAM. It showed that feature modelling can improve the maintainability of CAM thanks to available concise and scalable language like TVL. Moreover, a separation of concerns could be set and increased the accessibility of the system. This accessibility could also be increased through the clarifications of the interactions between model options offered by CAM what is the basic purpose of FMs. Visualization of these interactions is the key to primarily understand how configure a model. So the prototype implements a functionality to generate FDs from TVL models. As regards to the constant model evolution due to a constant domain evolution, increasing the maintainability through a more readable support like TVL models resolve part of the problem and version management could also be applied.

CAM showed up its use of default values to make the configuration process easier and light for users. This kind of structure is currently not part of FMs. So we decided to suggest their integration as it appeared to be a need in the industry. This integration was described in Chapter 6. We pointed out three types of default values: 1. initial configuration; 2. fill-in configuration; and 3. conditional default values. Then we defined an extension language of TVL to support conditional default values. However, this extension remains theoretical at this moment due to a lack of time, so no concrete implementation is provided yet.

Future Works

Our work showed the possibility to met the challenges offered by CAM in a restricted domain. Indeed, we had restrained the FM to Boolean options from CAM and their equivalent. Thus extend the model to support all types of options including numericals variables should be done. This extension will lead to the need to use a CSP solver inside the configurator instead of a SAT solver but the reasoning principles to manage configurations should remain the same.

Furthermore, at a first stage, our prototype had no configuration interface like a GUI and used an external tool, SPLOT, to represent it. The GUI of a configurator helps users to interactively configure a product. So implement a GUI to support configuration of TVL structure is another valuable future work. A possible lay out was presented in Section 5.2.2. This proposition is generic and so may not suit to end-users' needs as said A. Hubaux in his PhD thesis [19].

Finally, the last and not least future work is the implementation and test of the suggested language extension of TVL to support conditional default values. As the syntax and the semantics are already defined its integration should not be too difficult. We have just to keep in mind that default values even if they can be declaratively defined like constraints, they are more similar to user's choices. They are useful to automatically complete a configuration and should always be overridden by user's decisions when they involve the same variable.

Bibliography

[1] DIMACS CNF format. `http://logic.pdmi.ras.ru/~basolver/dimacs.html`.

[2] Research center in information system engineering. `http://www.fundp.ac.be/en/precise/`.

[3] Sameer Ansari. Reducing climate model configuration complexity using knowledge-based modeling. CS 4980 - Undergraduate Research Project at GeorgiaTech, College of Computing, May 2011.

[4] Wayne A. Babich. *Software configuration management: coordination for team productivity*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[5] Claire M. (Carlisle MA US) Plotkin Martin P. (Concord MA US) Colena Michael (Lowell MA US) Bagley. Interactive product configurator with default decisions. (20100037142), February 2010.

[6] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, September 2010.

[7] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated reasoning on feature models. In *LNCS, ADVANCED INFORMATION SYSTEMS ENGINEERING: 17TH INTERNATIONAL CONFERENCE, CAISE 2005*, page 2005. Springer, 2005.

[8] D. L. Berre, A. Parrain, O. Roussel, and L. Sais. Sat4j: A satisfiability library for java. `http://www.sat4j.org/`, 2005.

[9] Jeremy Blosser. Java tip 98: Reflect on the visitor design pattern. `http://www.javaworld.com/javaworld/javatips/jw-javatip98.html`, July 2000.

[10] Quentin Boucher, Andreas Classen, Paul Faber, and Patrick Heymans. Introducing TVL, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29*, pages 159–162. University of Duisburg-Essen, January 2010.

[11] John Burkardt. Satisfiability suggested format. `http://people.sc.fsu.edu/~jburkardt/pdf/dimacs_cnf.pdf`, May 1993.

[12] Andreas Classen, Quentin Boucher, Paul Faber, and Patrick Heymans. The TVL specification. Technical Report P-CS-TR SPLBT-00000003, PReCISE Research Center, University of Namur, Namur, Belgium, 2010.

[13] Krzysztof Czarnecki. *Generative programming : methods, tools, and applications*. Addison Wesley, Boston, 2000.

[14] Rocky Dunlap. Cesm configuration. `http://rockydunlap.wordpress.com/category/cesm/`.

[15] John Elison, Emden Gansner, Yifan Hu, and Arif Bilgin. The dot language. `http://www.graphviz.org/doc/info/lang.html`.

[16] John Elison, Emden Gansner, Yifan Hu, and Arif Bilgin. Graphviz - graph visualization software. `http://www.graphviz.org`.

[17] Paul Faber. Conception d'un logiciel automatisant le contrôle et l'analyse de modèles tvl. Master's thesis, FUNDP, 2010. Master's thesis.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[19] Arnaud Hubaux. *Feature-based configuration: collaborative, dependable, and controlled*. PhD thesis, FUNDP, 2012. PhD thesis.

[20] Arnaud Hubaux, Quentin Boucher, Herman Hartman, Rapaël Michel, and Patrick Heymans. Evaluating a text-based feature modelling language: Four industrial case studies (to appear). Accepted for publication at the 3rd International Conference on Software Language Engineering (SLE 2010), available online, July 2010.

[21] Arnaud Hubaux, D. Jannach, C. Drescher, L. Murta, T. Mannisto, Patrick Heymans, Krzysztof Czarnecki, T. Nguyen, and M. Zanker. Unifying Software and Product Configuration: A Research Roadmap. In *Proceedings of the Workshop on Configuration (ConfWS)*, Montpellier, France, 2012.

[22] L. Hvam, N.H. Mortensen, and J. Riis. *Product Customization*. Springer, 2008.

[23] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Software Engineering Institute, Carnegie Mellon University, November 1990.

[24] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[25] Christian Kästner, Sven Apel, and Klaus Ostermann. The road to feature modularity?, August 11.

[26] Charles W. Krueger. Introduction to software product lines. `http://www.softwareproductlines.com/introduction/introduction.html`.

[27] Marcilio Mendonca. *Efficient reasoning techniques for large scale feature models*. PhD thesis, University of Waterloo, 2009.

[28] Marcilio Mendonca, Moises Branco, and Donald Cowan. S.P.L.O.T. - software product lines online tools. `http://gsd.uwaterloo.ca:8088/SPLOT/articles/mendonca_splot_oopsla_2009.pdf`, October 2009.

[29] Marcilio Mendonca, Andrzej Wasowski, and Krzysztof Czarnecki. SAT-based analysis of feature models is easy. In *the 13th International Software Product Line Conference (SPLC'09)*, August 2009.

[30] Pedro J. Molina. Stringtemplate: a great template engine for code generation. `http://pjmolina.com/metalevel/2010/11/stringtemplate-a-great-template-engine-for-code-generation`, November 2010.

[31] NCAR. Community atmospheric model website. `http://www.cesm.ucar.edu/models/cesm1.0/cam/docs/ug5_1/ug.html`.

[32] NCAR. Community earth system model website. `http://www.cesm.ucar.edu/`.

[33] Jan Olhager. Strategic positioning of the order penetration point. *International Journal of Production Economics*, 85(3):319–329, September 2003.

[34] pure-systems GmbH. Variant management with pure::variant. `http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf`, 2006. Technical White Paper.

[35] Julio Sincero and Wolfgang Schröder-preikschat. The linux kernel configurator as a feature modeling tool, 2008.

[36] Peri Tarr, Harold Ossher, and William Harrison. N degrees of separation: Multi-dimensional separation of concerns. pages 107–119, 1999.

[37] Thomas Thum, Christian Kastner, Sebastian Erdweg, and Norbert Siegmund. Abstract features in feature modeling. In *Proceedings of the 2011 15th International Software Product Line Conference*, SPLC '11, pages 191–200, Washington, DC, USA, 2011. IEEE Computer Society.

[38] Thomas Thüm, Christian Kästner, Thomas Leich, Don Batory, et al. Featureide. `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`.

[39] Carnegie Mellon University. Software product lines. `http://www.sei.cmu.edu/productlines/`.

[40] Bruce Wallace. A time for reflection. `http://www.polyglotinc.com/reflection.html`.

[41] J. Whaley. The javabdd bdd library. `http://javabdd.sourceforge.net/`, 2003-2007.

Appendix A

Feature modelling of CAM

This annex presents the TVL model of CAM in standalone mode. This FM has been written according to the configuration script of CAM and the FD realized by R. Dunlap[1]. This FM only takes into account the climate model option that is used to be set with the CAM configuration script. It appears to be a basic FM with this limitation.

The TVL model is split into two files. The first file (Listing A.1 p.77) defines the features and the tree hierarchy between them. The second file (Listing A.2 p.79) defines additional constraints like cross-tree constraints.

```
root CAM
  group allof{
    opt UserCustomSource,
    opt Chemistry,
    Physics,
    Dynamics,
    Pbl,
    Interface,
    opt Parallelization,
    EsmfLibraryDirectory,
    RadiationDriver,
    opt Ocean,
    opt Land,
    opt SeaIce,
    opt CO2Cycle,
    Hgrid,
    opt PerturbationGrowthTests,
    opt Scam,
    opt Camiop
  }

Chemistry
  group allof{
    opt Chem_pkg,
    opt Prog_species
  }

  Chem_pkg
    group oneof{
      Super_fast_llnl,
      Super_fast_llnl_mam3,
      Trop_bam,
      Trop_ghg,
      Trop_mam3,
      Trop_mam7,
      Trop_mozart,
      Waccm_ghg,
      Waccm_mozart
    }

  Prog_species
    group someof{
      DST,
      SSLT,
      SO4,
      GHG,
      OC,
      BC,
      CARBON16
    }

Physics
  group allof{
    Phys_pkg,
    Radiation,
    Microphysics,
    opt Waccm
  }

  Phys_pkg
    group oneof{
      Cam3,
      Cam4,
      Cam5,
      Ideal,
      Adiabatic
    }

  Radiation
    group oneof{
      Rrtmg,
      Camrt
    }

  Microphysics
```

[1]Available at `http://rockydunlap.files.wordpress.com/2010/11/cam_features.pdf`

```
76      group oneof{
77        Mg,
78        Rk
79      }
80
81    Dynamics{
82      bool offline;
83      group oneof{
84        Eul,
85        Sld,
86        Fv,
87        Homme
88      }
89    }
90
91    Pbl
92      group oneof{
93        Uw,
94        Hb,
95        Hbr
96      }
97
98    Interface
99      group oneof{
100       Mct,
101       Esmf
102     }
103
104   Parallelization
105     group allof{
106       opt Spmd,
107       opt Smp
108       }
109
110   EsmfLibraryDirectory
111     group oneof{
112       Custom,
113       None
114     }
115
116   RadiationDriver
117     group oneof{
118       Offline,
119       Online
120     }
121
122   Ocean
123     group oneof{
124       Docn,
125       Dom,
126       Socn,
127       Aquaplanet
128     }
129
130   Land{
131     bool vocEmissions;
132     group oneof{
133       Clm,
134       Slnd
135     }
136   }
137
138   SeaIce
139     group oneof{
140       Cice,
141       Sice,
142       Csim4
143     }
144
145   Hgrid
146     group oneof{
147       EulGrid,
148       SldGrid,
149       FvGrid,
150       HommeGrid
151     }
152
153     EulGrid
154       group oneof{
155         E512x1024,
156         E256x512,
157         E128x256,
158         E64x128,
159         E48x96,
160         E32x64,
161         E8x16,
162         E1x1
163       }
164
165     SldGrid
166       group oneof{
167         S64x128,
168         S32x64,
169         S8x16
170       }
171
172     FvGrid
173       group oneof{
174         F0_23x0_31,
175         F0_47x0_63,
176         F0_5x0_625,
177         F0_9x1_25,
178         F1x1_25,
179         F1_9x2_5,
180         F2x2_5,
181         F2_5x3_33,
182         F4x5,
183         F10x15
184       }
185
186     HommeGrid
187       group oneof{
188         Ne2np4,
189         Ne7np8,
190         Ne5np8,
191         Ne10np4,
192         Ne16np4,
193         Ne16np8,
194         Ne21np4,
195         Ne30np8,
196         Ne60np4,
197         Ne120np4,
198         Ne240np4
199       }
200
201   /*Cross-tree constraints file*/
202   include(cam_CTconstraints_basic.tvl
        );
```

Listing A.1: TVL model of CAM in standalone mode

```
1  Chemistry{
2    !Chem_pkg && !Prog_species -> !this;
3  }
4
5  Prog_species{
6    this -> !Chem_pkg;
7  }
8
9  Physics{
10   (Phys_pkg.Ideal || Phys_pkg.Adiabatic)
11     -> !Chem_pkg;
12   (Phys_pkg.Cam3 || Phys_pkg.Cam4)
13     -> !(Chem_pkg.Super_fast_llnl_mam3 || Chem_pkg.Trop_mam3 || Chem_pkg
           .Trop_mam7);
14   Radiation.Camrt
15     -> !(Chem_pkg.Trop_mam3 || Chem_pkg.Trop_mam7 || Chem_pkg.
           Super_fast_llnl_mam3);
16   this.Radiation.Rrtmg ->  !Phys_pkg.Cam3 && this.Microphysics.Mg;
17   //this.Waccm -> Dynamics.Fv && (Chem_pkg || Prog_species);
18 }
19
20 Dynamics{
21   this.offline -> this.Fv;
22   this.Homme excludes SeaIce.Cice;
23   this.Eul || (PerturbationGrowthTests && this.Sld) <-> Hgrid.EulGrid;
24   this.Fv <-> Hgrid.FvGrid;
25   this.Sld <-> Hgrid.SldGrid;
26   this.Homme <-> Hgrid.HommeGrid;
27 }
28
29 Pbl{
30   this.Uw -> Physics.Microphysics.Mg;
31 }
32
33 RadiationDriver{
34   this.Online -> !Chem_pkg;
35 }
36
37 Land{
38   vocEmissions -> this.Clm;
39 }
40
41 Scam{
42   //this excludes Parallelization.Spmd;
43   this requires Dynamics.Eul;
44 }
45
46 Camiop{
47   this requires Dynamics.Eul;
48 }
```

Listing A.2: Additional constraints in TVL model of CAM

Appendix B

String patterns of CAMelot

Our prototype CAMelot uses a template engine called StringTemplate [30]. StringTemplate is a very powerful Java template engine. It allows the separation between the process and the output design. That means that we can define many different templates without changing anything in the process. It can be used for any formatted text output and uses an uncomplicated language to write the templates.

StringTemplate has its own syntax that is briefly described in Annex B.1. Annexes B.2 and B.3 shows in details the template used inside CAMelot to generate model into SXFM and DOT format respectively.

B.1 StringTemplate syntax: basics

The content of this section is extracted from the five minute introduction of StringTemplate available at `http://www.antlr.org/wiki/display/ST/Five+minute+Introduction`.

Note that:

1. each example in Table B.1 below fits on a single line but, it may appear on more than a line in the table due to space restrictions and wrapping.
2. StringTemplate supports the use of <...> and $...$ as delimiters (as shown in the examples)

Table B.1: Basic StringTemplate Syntax.

Syntax	**Example**
Description	
`<attribute>`	`$user$` `<user>`
Replaced with value of `attribute.ToString()` (or empty string if missing).	
`<attribute.property>`	`$user.name$` `<user.name>`
Replaced with value of `property` of `attribute` (or empty string if missing).	
`<attribute.(expr)>`	`$user.(name_label)$` `<user.(name_label)>`

Indirect property lookup. Same as `attribute.property` except value of `expr` is the property name.	
`<multi-valued-attribute>`	`$users$` `<users>`
Concatenation of `ToString()` invoked on each element.	
`<multi-value-attribute; separator=expr>`	`$users; separator=", "$` `<users; separator=", ">`
Concatenation of `ToString()` invoked on each element separated by `expr`.	
`<template(argument-list)>`	`$bold()$` `<bold(item=title)>`
Include (i.e. *call*) template. `argument-list` is a list of attribute assignments of the form *arg-of-template=expr*. *expr* is evaluated in the context of the surrounding template not of the invoked template.	
`<attribute:template(argument-list)>`	`$name:bold()$` `<name:checkoutReceipt(items=skus, ship=shipOpt)>`
Template *application*. The optional `argument-list` is evaluated before application. The default attribute it is set to the value of attribute. If `attribute` is multi-valued, it is set to each element in turn and `template` is invoked n times where n is the number of values in `attribute`.	
`<attribute:{argument-name_| _anonymous-template}>`	`$users:{s\|<li>$s$</li>}; separator="\n"$` `<users:{s|<li>$s$</li>}; separator="\n">`
Apply an anonymous template to each element of `attribute`. Set the `argument-name` to the iterated value and also set it.	
`<if(!attribute)>subtemplate<endif>`	`$if(users)$ $users:{u\|$u$}$ $endif$` `<if(users)> <users:{u|<u>}> <endif>`
If `attribute` has no value or is a *bool* object that evaluates to *false*, include `subtemplate`. These conditionals may be nested.	
`\$` or `\<`	`\$` `\<`
Escaped delimiter prevent $ or < from starting an attribute expression and results in that single character.	
`<\ >, <\n>, <\t>, <\r>`	`$\n$` `<\n>`
Special characters: space, newline, tab, carriage return.	
`<! comment !>, $! comment !$`	`$! this is a comment !$` `<! this is a comment !>`
Comments, ignored by StringTemplate.	

B.2 SXFM format

The SXFM format is a XML-based language used by SPLOT [28] to encode FM. SXFM supports basic FM structures that are *and*-, *or*- and *xor*-decompositions, mandatory and optional features, and cross-tree constraints in CNF.

The following template file (Listing B.1 p.82) in StringTemplate format groups a set of template required to write any basic FM in SXFM.

Line by line description of Listing B.1 p.82

Lines 1-2 init the template file and define the default delimiters for the template engine.

Lines 4-17 define *body* template. It represents the main structure of the XML file composed of *meta*, *feature_tree* and *constraints* tags inside the *feature_model* tag with the attribute *name*.

Lines 19-30 define the *meta* template. It lists all the meta data that could be provided for a FM store on the SPLOT platform.

Lines 32-34 define a generic *list* template.

Lines 36-39 define the *featureSet* template. It represents a group of feature with an Id, a cardinality constraint (min, max), and a set of features.

Lines 41-44 define the *feature* template. It represents a feature with its name, id, and list of children.

Lines 46-60 defines the four templates for the type of features that is root, mandatory, optional, and group feature.

Lines 62-72 define the templates to define constraints. The *constraint* template represent a constraint line equivalent to a clause. The *or* template represents the disjunction of a literal, and a list of literals. And the *not* template represents the negation of a literal.

```
1  group sxfm;
2  delimiters "$", "$"
3
4  body(name, meta, features, constraints) ::= <<
5  <!-- This model was created to be used by SPLOT's Feature Model Editor (
       http://www.splot-research.org) -->
6  <feature_model name="$name$">
7  <meta>
8  $meta$
9  </meta>
10 <feature_tree>
11 $features$
12 </feature_tree>
13 <constraints>
14 $constraints$
15 </constraints>
16 </feature_model>
17 >>
18
19 meta(description="", creator="", adresse="", email="", phone="", website
       ="", organization="", departement="", data="", reference="") ::= <<
20 <data name="description">$meta_description$</data>
21 <data name="creator">$meta_creator$</data>
22 <data name="address">$meta_adresse$</data>
```

```
23 <data name="email">$meta_email$</data>
24 <data name="phone">$meta_phone$</data>
25 <data name="website">$meta_website$</data>
26 <data name="organization">$meta_organization$</data>
27 <data name="department">$meta_departement$</data>
28 <data name="date">$meta_data$</data>
29 <data name="reference">$meta_reference$</data>
30 >>
31
32 list(item) ::= <<
33 $item; separator="\n"$
34 >>
35
36 featureSet(id, min, max, features) ::= <<
37 :g ($id$) [$min$,$max$]$if(features)$
38   $features$$endif$
39 >>
40
41 feature(name, id, children) ::= <<
42 $name$($id$)$if(children)$
43   $children$$endif$
44 >>
45
46 rootFeature(feature) ::= <<
47 :r $feature$
48 >>
49
50 mandFeature(feature) ::= <<
51 :m $feature$
52 >>
53
54 optFeature(feature) ::= <<
55 :o $feature$
56 >>
57
58 groupFeature(feature) ::= <<
59 : $feature$
60 >>
61
62 constraint(id, body) ::= <<
63 constraint_$id$:$body$
64 >>
65
66 or(expr1, expr2) ::= <<
67 $expr1$ or $expr2$
68 >>
69
70 not(expr) ::= <<
71 ~$expr$
72 >>
```

Listing B.1: Pattern of SXFM format

B.3 Feature diagram in DOT format

The DOT language is a text-based format to describe graphs. We used this language to provide a visualization to any TVL model. So, we have extended the FODA notation (Section 3.2) to integrate attributes in the FD. Attributes of a feature are inserted below the node of this feature with their type and id. The cross-tree constraints are inserted in the left-bottom corner, and so, don't really take part of the diagrams similarly to FD in FeatureIDE (Figure 4.2.2a). Figure B.3.1 represents the FD of the sample in TVL (Listing 5.2 p.54).

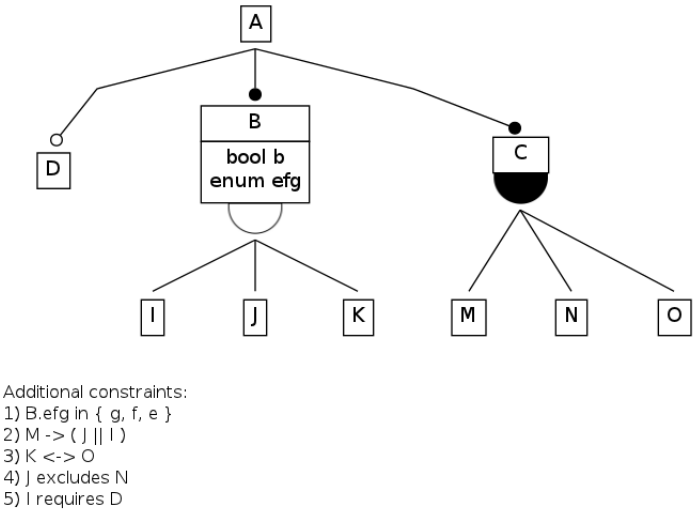


Figure B.3.1: FD example of a TVL model

The following template file (Listing B.2 p.86) in StringTemplate format groups a set of template required to generate a FD from any TVL model.

Line by line description of Listing B.2 p.86

Lines 1-2 init the template file and define the default delimiters for the template engine.

Lines 4-21 define the *diagram* template. It represents the main structure of the DOT file that is the diagram with its id, and parametrisation of the diagram.

Line 23 defines a generic *list* template.

Lines 25-27 define *or*- and *xor*-decompositions. Both require a trick to display their visual notation. We choose to embed a picture for each of those graphical notations. This picture is put inside the bottom cell of the table that represents the feature node in the graph (Lines 35).

Line 29 defines the *attribute* template. It represents an attributes like a couple of its type and Id.

Lines 31-44 define the *feature* template. It represents the feature as a node with its id shaped as a 3-cells table with two optional cells. The first mandatory cell hosts the feature name (Lines 32-33). The second cell is optional and hosts the feature attributes if there is at least one, otherwise the cell doesn't exist (Line 34). The

thirst cell also optional hosts the picture of the *or*-/*xor*-decomposition only if the children of the feature are constraint by such a decomposition, otherwise the cell doesn't exist (Line 35). Then the feature node is linked to its sub-feature node in a cluster.

Lines 47-53 define the two templates for the type of feature that is mandatory and optional respectively. These types are represented respectively with a filled circle and an empty circle. These templates are not used with features inside a *or*-/*xor*-decomposition.

Line 55 defines the constraint template. It represents the constraint like a couple of an id and its body that is its definition inside the normalized TVL model.

```
group FDdot;
delimiters "$", "$"

diagram(id, root, constraints) ::= <<
digraph $id$ {
splines=ortho;
compound=true;
concentrate=true;

node [shape=plaintext];
edge [headport=n, tailport=s, dir=both, arrowhead=none, arrowtail=none];

  $root$

$if(constraints.item)$
labeljust="l";
label="Additional constraints:\l$constraints.item; separator="\\l"$\l";
$endif$
fontsize=12;
}
>>

list(item) ::= "$item$"

orGroup(id) ::= "<IMG SRC=\"templates/or.png\"/>"

xorGroup(id) ::= "<IMG SRC=\"templates/xor.png\"/>"

attribute(id, type) ::="$type$ $id$"

feature(name, id, attributes, children, type, set) ::= <<
$id$[label=<<TABLE BORDER="0" CELLSPACING="0" CELLPADDING="4">
<TR><TD BORDER="1">$name$</TD></TR>
$if(attributes)$<TR><TD BORDER="1">$attributes.item; separator="<br />"$
    </TD></TR>$endif$
$if(set)$ <TR><TD BORBER="0" CELLPADDING="0">$set$</TD></TR>$endif$
</TABLE\>\>];
$if(children)$
subgraph cluster_$id$${
color=white;
$children.item: {c|$id$ -> $c.id$ $if(c.type)$$c.type$$endif$;$\n$}$
$children.item$
}
$endif$
>>


mandFeature(feature) ::= <<
[arrowhead=dot]
>>

optFeature(feature) ::= <<
[arrowhead=odot]
>>

constraint(id, body) ::= "$id$) $body$"
```

Listing B.2: Pattern of FD in DOT format